PROYECTO DE FIN DE CARRERA

TÍTULO DEL PFC: Adaptación de metodologías de Ingeniería de software orientadas a objeto al mantenimiento evolutivo de aplicaciones. Aplicación a un caso práctico.

TITULACIÓN: Ingeniería de Telecomunicaciones (segundo ciclo)

AUTOR: Daniel García Pino

DIRECTOR: Alfredo García Varela

SUPERVISOR: Antoni Oller Arcas

FECHA: 24-11-2008

Título: Adaptación de metodologías de Ingeniería de software orientadas a objeto al mantenimiento evolutivo de aplicaciones. Aplicación a un caso práctico.

Autor: Daniel García Pino

Director: Alfredo García Varela

Supervisor: Antoni Oller Arcas

Data: 24 de Noviembre de 2008

Resumen

El presente proyecto es un estudio acerca de metodologías de desarrollo de software. En concreto pretende diseñar una metodología adecuada para el supuesto que un determinado proyecto no parta de cero, es decir, que se base en la mejora/personalización de un software ya existente, o de la integración de varias aplicaciones.

Por otro lado, se exponen una serie de herramientas que pueden facilitar el seguimiento tanto de la metodología diseñada como de cualquier otra metodología.

Por último se evalúa la metodología aplicándola a un proyecto real de la empresa en la que se realiza este PFC.

Title: TFC/PFC Model

Author: Daniel García Pino

Director: Alfredo García Varela

Supervisor: Antoni Oller Arcas

Date: November, 24th 2008

Overview

The current Project is a study about software development methodologies. More concretely, it intends to design a methodology capable of carrying a special type of project, based on the improvement/customization of existing software, or the integration of various applications.

In the other hand, this project exposes some tools that can help the accomplishment of the designed methodology.

At last, the designed methodology is evaluated, putting it in practice with a real project.

ÍNDICE

INTRODUCCIÓN 1

CAPÍTULO 1. DESCRIPCIÓN DEL PROBLEMA 2

1.1 ANÁLISIS Y RECOGIDA DE REQUERIMIENTOS 2
1.2 APRENDIZAJE 3
1.3 DISEÑO 3
1.4 DESARROLLO 4

CAPÍTULO 2. METODOLOGÍAS DE DESARROLLO 5

2.1 WATERFALL 5
2.1.1 Filosofía 5
2.1.2 Proceso 5
2.1.3 Ventajas e inconvenientes 7
2.2 AGILE 8
2.2.1 Filosofía 9
2.2.2 Proceso 10
2.2.3 Ventajas e inconvenientes 15
2.3 SELECCIÓN DE LA METODOLOGÍA A SEGUIR EN EL CASO REAL 16

CAPÍTULO 3. HERRAMIENTAS DE DESARROLLO 20

3.1 HERRAMIENTAS DE DISEÑO 20
3.2 HERRAMIENTAS DE PROGRAMACIÓN 21
3.2.1 IDE 22
3.2.2 Compiladores 22
3.3 HERRAMIENTAS DE GESTIÓN DE LA DOCUMENTACIÓN 23
3.3.1 DokuWiki 24
3.3.2 Knowledge Tree 25
3.3.3 Alfresco 26
3.4 HERRAMIENTAS DE GESTIÓN DE PROYECTO 27
3.4.1 DotProject 28
3.4.2 Project Open 29
3.4.3 Microsoft Project 30
3.4.4 Open Workbench 31
3.5 HERRAMIENTAS DE TRABAJO EN EQUIPO 31
3.5.1 CVS 32
3.5.2 SVN 32
3.6 HERRAMIENTAS DE CONTROL DE CALIDAD 33
3.7 SELECCIÓN DE LAS HERRAMIENTAS A USAR EN EL CASO REAL 35

CAPÍTULO 4. CASO REAL 37

4.1 EL PROYECTO 37
4.1.1 Descripción 37
4.1.2 Requisitos 38
4.1.3 Decisiones iniciales 38
4.2 SUGARCRM 39
4.3 DESARROLLO DEL PROYECTO 41
4.3.1 Seguimiento 41
4.3.2 Diseño 43
4.3.3 Implementación 45
4.3.4 Resultado 48

CONCLUSIONES 51

BIBLIOGRAFÍA **53**
ANEXOS **54**
ANEXO A: DIAGRAMAS UML **54**

INTRODUCCIÓN

Desde la popularización de Internet y, sobre todo, con el nacimiento de Linux, el movimiento *open source* ha ido cobrando mayor importancia en el desarrollo de nuevas aplicaciones. Actualmente hay infinidad de software de libre distribución, creado por grandes comunidades de desarrolladores que aportan nuevas ideas desde distintos puntos de vista y, lo mejor de todo, es que se consiguen resultados tan buenos como los del software comercial.

El aspecto más interesante de este tipo de software (y lo que le da nombre) es que el código fuente de las aplicaciones se puede descargar libremente; esto permite modificar la aplicación como sea necesario para añadir o modificar la funcionalidad del programa y hacer que cumpla con unos requisitos determinados. De esta forma surge una nueva forma de desarrollar un proyecto de ingeniería de software, es decir, partiendo de una o varias aplicaciones *open source* que sirvan como base de una solución final.

El objetivo general de este proyecto es estudiar estas nuevas tendencias en el área de la ingeniería de software y, sobre todo, su aplicación en el aprovechamiento y evolución de software existente, para el desarrollo de nuevas aplicaciones. En esta línea, se estudiarán distintas herramientas y metodologías de desarrollo que facilitan el desarrollo de este tipo de proyectos.

Por otra parte, el objetivo principal de este proyecto es definir una metodología de desarrollo que permita trabajar de forma diligente con este tipo de proyectos. Esta metodología se personalizará para un proyecto en concreto, y se evaluará aplicándola a dicho proyecto.

El proyecto está estructurado en dos partes: La primera de ellas, comprendida entre los capítulos del 1 al 3, se centra en el estudio teórico (aunque supondrá distintas pruebas, sobre todo en el capítulo de herramientas) de las metodologías y herramientas que se pueden aplicar al tipo de proyectos que vamos a estudiar. Lo más importante de esta parte es la definición de la metodología personalizada, mencionada anteriormente.

La segunda parte, que se corresponde con el capítulo 4, se trata del desarrollo de un caso real usando las técnicas expuestas (principalmente siguiendo la metodología personalizada) en la parte teórica. El caso trata de la adaptación de un software de gestión de clientes (CRM) para satisfacer las necesidades de un cliente. Con esta segunda parte se podrá evaluar de forma realista el comportamiento de la metodología diseñada.

CAPÍTULO 1. DESCRIPCIÓN DEL PROBLEMA

Habitualmente, las metodologías de ingeniería de software parten de que no hay ninguna aplicación previa al desarrollo del proyecto, es decir, se piensa y se diseña la aplicación teniendo en cuenta que todo el código que la componga será nuevo. Esta aproximación al desarrollo de aplicaciones es mucho más libre que la que se pretende plantear en este proyecto, ya que permite diseñar totalmente la estructura de la aplicación.

El problema es que, en un sector tan competitivo, el empezar de cero puede hacer que se pierda mucho tiempo y posibilidades de negocio. Por ello, y cada vez más, actualmente lo más normal cuando se empieza un proyecto es que ya se tiene algo (ya sea una o varias aplicaciones a modificar, un determinado *framework*, o una versión anterior del software).

Lo que ocupa al presente proyecto son las diferencias que surgen en el desarrollo del proyecto si se tiene en cuenta que ya hay parte de la aplicación desarrollada. En los subapartados de este capítulo se irán definiendo los problemas o diferencias que pueden surgir en cada etapa de un proyecto de mejora o adaptación de software.

1.1 Análisis y recogida de requerimientos

La etapa de análisis y recogida de requerimientos tiene como finalidad analizar las necesidades del cliente, para luego poder determinar distintos aspectos del software que se va a desarrollar. Estos aspectos pueden ir desde estimar los recursos requeridos, empezar a tomar decisiones de diseño o, teniendo en cuenta la temática del presente proyecto, decidir si se va a desarrollar el software de cero o si se aprovechara la funcionalidad de una aplicación existente. Pero al final lo más importante de esta etapa es estimar el coste económico que tendrá el proyecto, puesto que esto suele ser lo que más le interesa al cliente.

De cara al equipo de desarrollo, lo más importante de la etapa de análisis es comprender la situación de la empresa, estimar los conocimientos técnicos que tiene el personal de la misma, analizar sus necesidades y, en general, cualquier aspecto que puedan ayudar a la hora de tomar las decisiones iniciales del proyecto.

Por otro lado, y como ya se ha dicho, es en esta etapa en la que se decide si tomar el camino de usar un software existente o no, por lo tanto, también es el momento de decidir la metodología que se va a seguir (ayudándonos de toda la información recopilada acerca del cliente). Por este motivo, esta etapa de análisis no difiere en exceso de una metodología a otra, porque se hace en un momento en que aun no se ha decidido la metodología a seguir.

En cambio, lo que si se va a ver afectado es la etapa de recogida de requerimientos, ya que hay que seguir una pautas para ello dependiendo de la metodología elegida (ver **Capítulo 2**). Aun así, este cambio se puede deber más al tipo de metodología elegida que al hecho de partir de un software existente o no.

1.2 Aprendizaje

Entre la etapa de análisis y la de diseño, se añade una nueva etapa de aprendizaje, en la cual se debe estudiar cómo está estructurada la aplicación (o aplicaciones) de la que se parte y, en el caso de no haber trabajado nunca con él, aprender el lenguaje de programación con el cual se ha desarrollado dicha aplicación. Es importante saber si la aplicación tiene mecanismos (que permita la instalación de módulos personalizados o la sustitución de parte del código) que faciliten la tarea de desarrollar nuevas funcionalidades, o si estas se tienen que desarrollar en forma de programa externo. Asimismo hay que ver que partes se tienen que programar y cuales se pueden solucionar mediante configuración de la aplicación.

Por otro lado, es importante determinar hasta qué punto la funcionalidad heredada de la aplicación base es importante en el proyecto, puesto que esta etapa de aprendizaje hace que se incrementen los costes de desarrollo. Si se gastan muchos recursos en la etapa de aprendizaje para luego poder aprovechar una funcionalidad concreta de la aplicación base, el coste añadido no se compensa con un ahorro posterior a la hora de implementar la funcionalidad del sistema.

Resumiendo, se requiere un estudio intensivo de la aplicación de la que se parte, de forma que se pueda adquirir los conocimientos necesarios para poder desarrollar las nuevas funcionalidades requeridas, a la vez que se evalúa la reusabilidad de dicha aplicación dentro del marco del proyecto en curso.

1.3 Diseño

Una vez pasada la etapa de aprendizaje se puede pasar a la etapa de diseño, pero teniendo en cuenta que el esqueleto de la aplicación ya está hecho y que no se puede cambiar. Por ello, todo el diseño se ha de adaptar a dicho esqueleto. Además de esto, se ha de pensar como los nuevos elementos añadidos a la aplicación interactuarán con todo lo existente.

Por otro lado, en el caso de usar distintas piezas de software, o desarrollar parte del proyecto como una aplicación externa hay que diseñar las distintas interfaces que permitirán la integración de unas piezas con otras.

Teniendo en cuenta todo lo dicho, es evidente que se necesitan herramientas que nos permitan obtener todo lo necesario para comprender en profundidad

cómo funciona la aplicación base. Ejemplos del tipo de herramientas que necesitamos son aquellas que nos permiten hacer ingeniería inversa, diagramas UML, etc. De todo esto se habla más adelante en el capítulo dedicado a herramientas de desarrollo.

1.4 Desarrollo

Al partir de una aplicación existente, de la cual se tendrá que modificar su código fuente, a veces puede ser difícil determinar en qué parte del código se ha introducido un cambio, lo cual puede dificultar la implementación de futuras funcionalidades y el mantenimiento de la aplicación. Por ello, en la etapa de desarrollo, lo más importante es encontrar una forma de ir controlando cada cambio que se realice en la aplicación. Es importante identificar los distintos tipos de cambios que se pueden realizar a la aplicación base, para así facilitar la posterior documentación de los mismos.

CAPÍTULO 2. METODOLOGÍAS DE DESARROLLO

Para afrontar los problemas planteados en el capítulo anterior, se están empezando a usar nuevas metodologías de desarrollo que se engloban bajo el nombre de metodologías ágiles. En este capítulo se describen estas metodologías, pero no sin antes describir la metodología usada tradicionalmente (*waterfall* [1]), con el objetivo de comparar las ventajas e inconvenientes de cada una.

2.1 Waterfall

Como ya se ha dicho, en este apartado se expone la metodología clásica de desarrollo, ampliamente usada desde el nacimiento de la ingeniería de software hasta la actualidad. Básicamente se describirá por encima la filosofía asociada a esta metodología, los pasos que se siguen con el objetivo de llevar el proyecto a buen puerto y las ventajas e inconvenientes que plantea esta metodología.

2.1.1 Filosofía

Tal y como su nombre indica, la metodología *waterfall* está basada en una filosofía lineal (*waterfall*, en inglés, significa cascada). En ella se ve el proyecto como una serie de pasos que hay que seguir en un orden fijo e invariable. La principal preocupación del equipo de desarrollo es que el proyecto mantenga una estructura fija y muy bien definida, lo cual se consigue documentando cada paso que se da. De esta forma cada decisión tomada se debe aprobar y, salvo en casos excepcionales, se mantiene invariable en todas las etapas posteriores del proyecto.

Este tipo de pensamiento viene derivado de las metodologías empleadas en el desarrollo de proyectos de ingeniería industrial (a causa de que los primeros proyectos de ingeniería de software se realizaron en entornos industriales), en los cuales la retroalimentación entre las diferentes etapas era prácticamente nula y el proceso requería una estructura rígida e invariable.

Por todo lo dicho, se puede resumir los principios de esta metodología en tres conceptos: rigidez, linealidad y orden.

2.1.2 Proceso

En este apartado, se definen los distintos pasos (o distintas etapas) que se deben seguir para la finalización de un proyecto mediante el método *waterfall*.

Hay que tener en cuenta que el esquema definido se corresponde con un esquema genérico y que está totalmente abierto a modificaciones.

Las distintas etapas por las que pasa un desarrollo siguiendo esta metodología son las siguientes:

- **Análisis y definición de requerimientos:** Los servicios, restricciones y metas del sistema se definen a partir de las consultas con los usuarios. Entonces, se definen en detalle y sirven como una especificación del sistema. Habitualmente esta especificación se mantiene invariable hasta el final del desarrollo.

- **Diseño del sistema y del software:** El proceso de diseño del sistema divide los requerimientos en sistemas hardware o software. Establece una arquitectura completa del sistema. El diseño del software identifica y describe las abstracciones fundamentales del sistema software y sus relaciones.

- **Implementación y prueba de unidades:** Durante esta etapa, el diseño del software se programa como un conjunto de subsistemas de software. Cada subsistema se debe testear por separado para asegurar que cumpla su especificación.

- **Integración y prueba del sistema:** Los subsistemas de software se integran y se prueban como un sistema completo para asegurar que se cumplan los requerimientos del software. Después de las pruebas, el sistema software se entrega al cliente.

- **Funcionamiento y mantenimiento:** Por lo general (aunque no necesariamente), ésta es la fase más larga del ciclo de vida. El sistema se instala y se pone en funcionamiento. El mantenimiento implica corregir errores no descubiertos en las etapas anteriores del ciclo de vida, mejorar la implementación de las unidades del sistema e implementar nuevas características una vez que se descubren nuevos requerimientos.

En la siguiente figura se puede ver un esquema del proceso descrito:

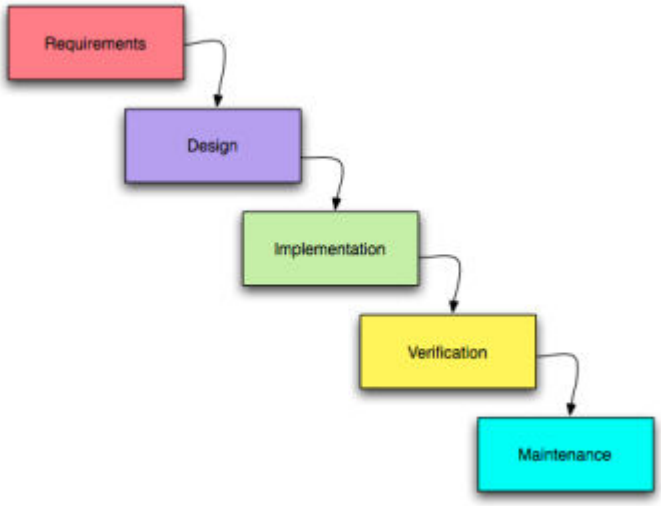

Fig. 2.1 esquema de la metodología *Waterfall*

En principio, el resultado de cada fase es uno o más documentos aprobados, y la siguiente fase no debe empezar hasta que la fase previa haya finalizado. En la práctica, estas etapas se superponen y proporcionan información a las otras, por ejemplo, durante el diseño se identifican problemas con los requerimientos; durante la implementación se descubren errores de diseño, y así sucesivamente.

El proceso de desarrollo de un proyecto de ingeniería de software no es un modelo lineal simple, sino que implica una serie de iteraciones de las diferentes etapas. Aun así, debido a los costos de producción y aprobación de documentos, las iteraciones son costosas e implican rehacer trabajo. Por ello, lo normal es que tras unas pocas iteraciones se detengan ciertas etapas del desarrollo (como puede ser la especificación de requerimientos) para poder continuar con las siguientes. Esto último provoca que se deban pasar por alto algunos problemas que puedan surgir en etapas posteriores, con lo cual, al final, podríamos obtener un software mal estructurado (debido a errores de diseño solucionados mediante trucos de implementación) o que no hace lo que el cliente quiere que haga (como consecuencia de la omisión de requerimientos).

2.1.3 Ventajas e inconvenientes

El punto fuerte de esta metodología se encuentra en el orden. Al tener una estructura fija e invariable, permite mantener una muy buena gestión de las diferentes tareas que componen el proyecto. Asimismo, es más fácil hacer previsiones acertadas en los plazos de entrega y controlar el progreso del proyecto (gracias a que se va documentando todo lo que se hace).

A su vez, este orden también supone su punto débil. La inflexibilidad en los procesos y en las decisiones puede suponer consecuencias desastrosas en el

producto final. El principal problema de la inflexibilidad se encuentra en la prácticamente nula retroalimentación de los procesos. Las reuniones con los clientes no sirven para nada más que discutir el progreso del proyecto, nunca para probar las características ya implementadas, con lo cual no puede opinar y pedir cambios en el funcionamiento de la aplicación. Esto hace que un error de entendimiento de los requisitos pueda llevar a un producto final insatisfactorio de cara al cliente.

Por otro lado, el obstáculo de esta metodología se encontró al quererla adaptar a equipos de desarrollo pequeños. En estos casos, la rigidez y, sobre todo, la continua y costosa documentación hace que se dificulte el continuo avance del proyecto. Además, si se diese el caso de no satisfacer algunos requisitos del cliente, rehacer el trabajo con pocos desarrolladores puede resultar una tarea muy complicada.

2.2 Agile

En los años 80 y principios de los 90, existía una opinión general de que la mejor forma de obtener un mejor software era a través de una planificación cuidadosa del proyecto, una garantía de calidad formalizada y procesos de desarrollo de software controlados y rigurosos, es decir, la metodología *Waterfall*. Esta opinión provenía, fundamentalmente, de la comunidad de ingenieros de software implicada en el desarrollo de grandes sistemas software de larga vida que normalmente se componían de un gran número de programas individuales.

Este software era desarrollado por grandes equipos que a veces trabajaban para compañías diferentes. A menudo estaban dispersos geográficamente y trabajaban en el software durante largos periodos de tiempo. Estos enfoques implican una importante sobrecarga de trabajo en cuanto a la planificación, diseño y documentación del sistema. Este esfuerzo adicional se justifica cuando se tiene que coordinar el trabajo de múltiples equipos de desarrollo, cuando el sistema es un sistema crítico y cuando muchas personas diferentes estarán involucradas en el mantenimiento del software durante su vida.

Sin embargo, cuando este enfoque «pesado» de desarrollo basado en la planificación fue aplicado a equipos de desarrollo pequeños y medianos, el esfuerzo invertido en documentación y planificación era tan grande que algunas veces dominaba el proceso de desarrollo del software. Se pasaba más tiempo pensando en cómo se debía desarrollar el sistema que en programarlo y probarlo. Además, cuando cambiaban los requerimientos, se hacía esencial rehacer todo el trabajo y la especificación y el diseño tenían que cambiar con el programa. Todo ello dificulta el avance del proyecto ya que, al tratarse de equipos de desarrollo pequeños, no se puede repartir demasiado el trabajo.

El descontento con estos enfoques pesados condujo a varios desarrolladores de software en los años 90 a proponer los nuevos métodos ágiles. Éstos permitieron a los equipos de desarrollo centrarse en el software mismo en vez

de en su diseño y documentación. Los métodos ágiles universalmente dependen de un enfoque iterativo para la especificación, desarrollo y entrega del software, y principalmente fueron diseñados para apoyar el desarrollo de aplicaciones de negocio donde los requerimientos del sistema cambiaban rápidamente durante el proceso de desarrollo. Están pensados para entregar software funcional de forma rápida a los clientes, quienes pueden entonces proponer que se incluyan en iteraciones posteriores del sistema nuevos requerimientos o cambios en los mismos.

En este apartado, siguiendo la misma estructura que para el caso de la metodología *Waterfall*, se exponen los distintos aspectos que definen este tipo de metodologías.

2.2.1 Filosofía

La filosofía asociada a estas metodologías se recoge en el manifiesto Agile [3]. Este manifiesto se compone de 12 principios básicos, a partir de los cuales los desarrolladores que quieran seguir esta filosofía deben diseñar sus propias metodologías. De esta forma, distintos autores ya han escrito sobre distintas metodologías que intentan seguir estos principios. Los 12 principios son los siguientes:

1. La principal prioridad es satisfacer al cliente mediante entregas tempranas y continuas de software útil.

2. Los cambios en los requerimientos son bienvenidos, incluso en las etapas finales del desarrollo. Los procesos ágiles se aprovechan de los cambios para mejorar la competitividad del cliente.

3. Se deben hacer entregas de software funcional de forma frecuente, desde pocas semanas hasta pocos meses entre entregas, con preferencia para las escalas de tiempo más pequeñas.

4. El personal del cliente debe colaborar con los desarrolladores diariamente con el fin de avanzar en el desarrollo del proyecto.

5. Los proyectos deben ser desarrollados por individuos motivados. Para ello se les debe proporcionar el entorno y el soporte que necesitan, además de confiar plenamente en cada uno de ellos.

6. La forma más eficiente y efectiva de transmitir información entre los miembros de un equipo de desarrollo es la conversación cara a cara.

7. El software funcional es la principal medida del progreso.

8. Los procesos ágiles promueven el desarrollo sostenible. Los patrocinadores, desarrolladores y usuarios deben ser capaces de mantener un ritmo constante de forma indefinida.

9. La atención continua a la excelencia técnica y al buen diseño mejora la agilidad.

10. La simplicidad, entendida como el arte de maximizar la cantidad de trabajo no realizado, es esencial.

11. Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.

12. A intervalos regulares, el equipo refleja como volverse más efectivo, entonces modifica su comportamiento para conseguirlo.

De estos principios que conforman el manifiesto agile se destacan por encima de todo dos aspectos: el primero es que todos los procesos ágiles se basan en entregas incrementales, es decir, pequeñas versiones del software que implementan parte de los requerimientos; el segundo aspecto importante es la colaboración constante del cliente (algo que es tan fundamental como, a veces, complicado de llevar a cabo).

Como se puede ver, la filosofía asociada a este tipo de metodologías es mucho menos rígida, basada en la comunicación directa con el cliente y, sobre todo (como su nombre indica), en la agilidad de los procedimientos (ya que permiten ir hacia adelante o hacia atrás hasta satisfacer al cliente).

2.2.2 Proceso

En este caso, la filosofía que se quiere imponer no ha dado lugar a una sola metodología, sino que a partir de los principios del *agile manifestó* han surgido una serie de procedimientos distintos. Las metodologías más famosas son: *Scrum*, *eXtreme Programming (XP)* y *Agile Unified Method*. De estas tres las más usadas son *Scrum* y *XP* [1], por ello, estas dos metodologías son las que se van a describir en este apartado.

2.2.2.1 Scrum

Como ya se ha dicho, *Scrum* es una metodología para la gestión de proyectos englobada dentro del ámbito de las metodologías ágiles. Esta metodología fue expuesta por Hirotaka Takeuchi e Ikujiro Nonaka, en el artículo *The New Product Development Game* (*Harvard Business Review*, Ene-Feb. 1986) en el que ponen de manifiesto que:

- El mercado competitivo de los productos tecnológicos, además de los conceptos básicos de calidad, coste y diferenciación, exige también rapidez y flexibilidad.

- Los nuevos productos representan cada vez un porcentaje más importante en el volumen de negocio de las empresas.

- El mercado exige ciclos de desarrollo más cortos.

El artículo compara el desarrollo tradicional de ciclo de vida formado por fases separadas y equipos especializados con las carreras de relevos, donde un equipo pasa el testigo al siguiente hasta finalizar el desarrollo del producto. Siguiendo el símil deportivo, se compara al nuevo modelo de desarrollo, basado en el solapamiento de las fases y en un único equipo multi-disciplinar, con la evolución del juego del rugby; y de él se toma el término *Scrum* (que se trata de una táctica comúnmente usada en este deporte).

De forma resumida, *Scrum* distingue entre dos elementos principales: Actores y acciones. Los actores son las personas que ejecutan las acciones, y las acciones son las distintas fases del ciclo de desarrollo de *Scrum*. En cuanto a los actores se pueden diferenciar cuatro roles:

- ***Product Owner*:** Es el responsable del proyecto por parte del cliente. Se encarga de definir las prioridades en las tareas a realizar, evaluar el avance del proyecto desde el punto de vista del cliente y, en general, servir de punto de comunicación entre el cliente y el equipo de desarrollo.

- ***Scrum Master*:** Es la persona que se encarga de que se cumplan los principios de la metodología, así como servir de guía para los integrantes del equipo a través de reuniones.

- ***Scrum Team*:** Son el equipo de desarrollo del proyecto. Se encargan de implementar las funciones requeridas por el *Product Owner*.

- **Usuarios:** Son los usuarios finales de la aplicación. A partir del progreso de la aplicación, pueden aportar nuevas ideas de modo que la aplicación final se adapte a sus necesidades personales.

Por otro lado, las acciones de *Scrum* consisten en una serie de fases del ciclo iterativo que sigue la metodología. Las acciones principales de *Scrum* son las siguientes:

- ***Product Backlog*:** Es una lista de todas las tareas, funcionalidades o requerimientos del proyecto. El *Product Owner* se encarga de priorizar estas tareas y de actualizar la lista con los objetivos conseguidos.

- ***Sprint Backlog*:** Es un periodo de entre una y cuatro semanas en el cual se deben realizar una serie de tareas sacadas del *Product Backlog*. La duración y las tareas a realizar en el *Sprint Backlog* se deciden antes del mismo, y se mantienen invariables hasta la finalización del mismo.

- ***Daily Scrum Meeting*:** Es una reunión diaria, entre los miembros del equipo, que se realiza durante el tiempo que dura el *Sprint*. Debe

tratarse de una reunión informal y ágil, de no más de 30 minutos, y en la cual se le plantean las 3 siguientes preguntas a los miembros del equipo:

- ¿Qué has hecho desde la última reunión?
- ¿Qué vas a hacer hoy?
- ¿Qué obstáculos se te presentan? Tras la respuesta a esta pregunta debe actuar el *Scrum Master*, que se encargará de eliminar dichos obstáculos.

Además de estas acciones, se pueden encontrar otras secundarias que ayudan a mantener un avance ordenado del proyecto, así como permitir que el cliente (o *Product Owner*) pueda ir viendo este avance. Dichas acciones son las siguientes:

- ***Sprint Planning Meeting*:** es una reunión, situada entre el *Product Backlog* y el *Sprint Backlog*, que tiene como finalidad el mover las tareas del *Product Backlog* al *Sprint Backlog*, es decir, decidir las tareas que se realizarán en el siguiente *sprint*. En este tipo de reuniones suelen participar el *Product Owner* (que es quien prioriza las tareas), el *Scrum Master* y el *Scrum Team*. De esta reunión también sale el *Sprint Goal*, que es un pequeño documento explicativo acerca de donde se quiere llegar con el siguiente *Sprint*.

- ***Sprint Review*:** una vez finalizado un *sprint*, se debe tener alguna parte nueva del software funcionando, para poder mostrársela al cliente. El objetivo del *Sprint Review* es mostrar los avances conseguidos durante el periodo del *Sprint*.

- ***Sprint Retrospective*:** Una vez concluido el *Sprint Review, el Scrum Master* revisará con el equipo los objetivos marcados en el *Sprint Backlog* recién finalizado, se analizarán los aspectos positivos y negativos de *Sprint* y, acorde con este análisis, se realizarán cambios y ajustes en la planificación del proyecto.

En el siguiente diagrama se puede ver todo el proceso completo de esta metodología.

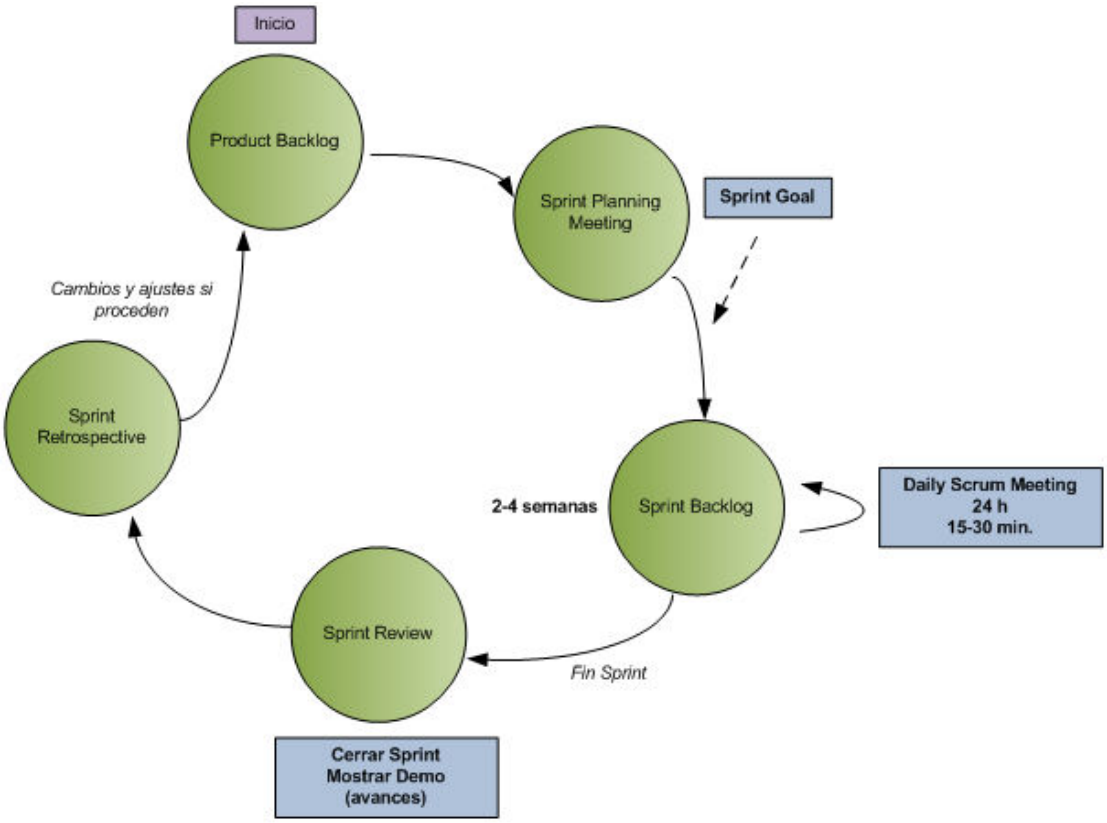

Fig. 2.2 Esquema del proceso de *Scrum*

2.2.2.2 eXtreme Programming

Esta metodología fue formulada de forma pública por Kent Beck, el cual escribió el primer libro sobre la materia (*Extreme Programming Explained: Embrace Change*, 1999). Se considera una de las metodologías ágiles más utilizadas y, por ello, se ha estimado hacer un resumen de la misma en este capítulo.

Las características principales de la programación extrema (de ahora en adelante XP) son las siguientes:

- **Desarrollo iterativo e incremental**.

- Los **requerimientos** se expresan como escenarios (también conocidos como historias de usuarios), los cuales se dividen e implementan como una serie de tareas.

- Se escriben **pruebas unitarias** para cada tarea. No se considerará una unidad de código como correcto hasta que no pase todos los test escritos. Se recomienda escribir las pruebas antes que el código de la tarea.

- Los desarrolladores **trabajan en parejas**. Se da prioridad a un código de mayor calidad, discutido y revisado desde dos puntos de vista distintos, que a una mayor productividad (lo normal es que dos personas programando juntas sean más lentas que una sola).

- **Integración** del equipo de programación con el cliente. Se recomienda que un representante del cliente trabaje de forma conjunta con el equipo de desarrollo.

- **Refactorización del código**, es decir, reescribir parte del código para aumentar su legibilidad, mantenibilidad y capacidad de reutilización.

- **Propiedad compartida del código**, es decir, que la responsabilidad del desarrollo de cada módulo se divide entre varios grupos de trabajo, de forma que cualquier persona del equipo pueda modificar e introducir mejoras en cualquier parte. Gracias a esto la mayoría de errores se acaban detectando.

- **Simplicidad** en el código. La mejor manera de que algo funcione es que sea simple, al menos en su inicio. Más adelante, cuando todo funciona bien, se pueden añadir funcionalidades más complejas.

- **Comunicación**. Es una premisa de la filosofía de desarrollo ágil, una buena comunicación entre los integrantes de un equipo de desarrollo evita malentendidos que puedan afectar posteriormente al desarrollo del proyecto.

Los pasos a seguir en la XP serían los mismos que en cualquier metodología basada en el desarrollo iterativo, con entregas cada dos meses aproximadamente. Sin embargo, de esta metodología hay que destacar una serie de peculiaridades que la diferencian del resto de metodologías.

En la XP, los procesos empiezan con la definición de las “tarjetas de historias”. Estas tarjetas se deben definir de forma conjunta con el cliente, ya que este debe especificar en ellas historias de la vida cotidiana de la empresa que se quieren solventar con la aplicación. Así se consigue determinar las necesidades del cliente. A partir de las tarjetas de historias se definen una serie de tareas y se estima el esfuerzo y los recursos requeridos para su implementación. Tras esto el cliente debe priorizar las historias que piensa que podrán ser útiles de forma inmediata. Se puede ver, ya en este proceso inicial, la importancia que tiene la participación del cliente. El mismo también tiene que tomar parte activa cuando hay un cambio en los requerimientos de la aplicación, desarrollando nuevas tarjetas de historias y decidiendo si van primero los cambios o las nuevas funcionalidades.

Centrándose más en aspectos del desarrollo, la XP, como su nombre indica, adopta un enfoque “extremo” para el desarrollo iterativo. Se puede llegar a construir varias versiones del software en un mismo día, siendo obligatorio que cada nueva versión pase todos los test unitarios, y no sólo los de las nuevas funcionalidades. Por otro lado, la XP también pone especial énfasis en la

refactorización del código, de forma que el software se adapte más fácilmente (consiguiendo que el código sea más claro, estructurado y reusable) a los cambios imprevistos que puedan surgir en el futuro.

Finalmente, hay que destacar otra peculiaridad que introducen los procesos basados en la XP: el trabajo en parejas. Esta medida tiene 3 objetivos principales: el primero es repartir la propiedad y responsabilidad del código entre todos los integrantes del equipo de desarrollo, de forma que tanto los meritos como los errores no se atribuyan a alguien de forma individual; el segundo es introducir un proceso informal de revisión del código, ya que cada línea de código la han revisado como mínimo dos personas; el último objetivo es ayudar a la refactorización del código, puesto que la misma presenta un beneficio inmediato al resto de integrantes del equipo, algo que en entornos de desarrollo tradicionales se ve como un beneficio a largo plazo.

2.2.3 Ventajas e inconvenientes

La principal ventaja de este tipo de metodologías es, como su nombre indica, la agilidad de los procesos. Puesto que cada iteración se pasa por todas las etapas de desarrollo clásicas, cada característica añadida se estudia de una forma menos generalista con lo cual, habitualmente, se llega a una mejor implementación de cada una de ellas. En la misma línea, gracias a la continua participación del cliente (el cual valida el trabajo realizado en cada iteración), se obtiene una mayor facilidad para que la aplicación se ajuste al máximo a lo que se necesita.

Por otro lado, el principal punto negativo se encuentra en el hecho de que es fácil romper la estructura del proyecto y que lleguen momentos en que no se sabe exactamente todo lo que se ha hecho. Esto se debe a la falta de documentación que pueden tener los proyectos de este tipo. El principal elemento para medir el avance del proyecto se trata de código funcional, con lo tanto se incita a los desarrolladores a centrarse en programar cada característica (y más si se tiene en cuenta los cortos plazos que propone cada iteración) y ello puede ocasionar que los mismos desarrolladores no tengan tiempo para documentar el trabajo realizado.

Por todo ello, este tipo de metodologías es recomendable en los casos en que los requisitos no están claros o pueden cambiar con frecuencia (algo que pasa habitualmente en proyectos pequeños en los cuales el cliente no sabe exactamente lo que quiere). También hay que tener en cuenta que los equipos de desarrollo no pueden ser de más de 10 personas puesto que la principal vía de comunicación entre los miembros del equipo es la comunicación cara a cara, y si el equipo es muy grande esto puede ocasionar problemas y malentendidos.

Resumiendo, este tipo de metodologías son adecuadas para equipos de desarrollo pequeños y para proyectos que no traten la creación de sistemas críticos o de gran escala, ya que en sistemas de mayor envergadura el

descontrol y la falta de documentación pueden hacer que se pierda por completo el rumbo.

2.3 Selección de la metodología a seguir en el caso real

En el proyecto que nos ocupa, y dado las circunstancias de la empresa en que se desarrolla el mismo, se usará una metodología que combina parte de las tres descritas, aunque se pretende hacer mayor hincapié en las metodologías ágiles, ya que se adaptan más a las características del proyecto. Por otro lado, hay que destacar que ninguna de las metodologías ágiles descritas puede ser usada por completo en este caso particular por motivos relacionados con la organización de la empresa y, por tanto, la metodología se ha tenido que "personalizar".

La peculiaridad principal del equipo de desarrollo de este proyecto es que está formado por un director de proyecto y un sólo desarrollador (aunque actualmente el equipo ya dispone de dos desarrolladores), el cual trabaja a distancia. Por ello, una de las premisas de los desarrollos ágiles no se puede cumplir al 100%, y es el hecho de que el equipo de desarrollo tiene que colaborar directamente (a poder ser cara a cara) con el cliente. En este caso se ha optado a que sólo asista a las sesiones de colaboración con el cliente el director del proyecto y, en el caso que fuera necesario, el desarrollador se comunica con el cliente por teléfono.

También son importantes en la elección de la metodología las características del cliente para el cual se va a realizar el proyecto. El aspecto más importante del mismo es la falta de conocimientos técnicos. Esto provoca que el cliente no tenga demasiado claro lo que quiere ni como lo quiere, con lo cual es más adecuado decantarse hacia las metodologías ágiles porque seguramente los requisitos serán muy cambiantes y porque, de esta forma, se puede ir orientando al cliente para que descubra lo que necesita realmente.

Por otro lado, hay que tener en cuenta que el objetivo es idear una metodología que permita trabajar con cierto ritmo a partir de un software ya existente, lo cual supone la peculiaridad principal que se quiere tratar en este proyecto. En los siguientes párrafos se expone la metodología diseñada para el presente proyecto, así como las medidas que hay que tomar para aprovechar al máximo el software del que se parte.

El proceso definido empieza con una reunión con el cliente, en la cual se intentaran definir los requisitos generales, a la vez que se recoge información para poder hacer una valoración económica del posterior desarrollo. Para ello se discutirá sobre el funcionamiento general de la empresa del cliente, los distintos departamentos que la componen, los tipos de empleados, etc. Con esto se podrán definir los actores de la aplicación, es decir, las personas que usaran la misma. Otro dato interesante es que el equipo entienda la actividad de cada departamento (y de sus integrantes) en la empresa, para así poder estimar lo que va a necesitar cada uno dentro de la aplicación. De la misma

forma, se expresará al cliente que el desarrollo se realizará por departamentos y se le pedirá que establezca unas prioridades (de forma que pueda tener operativa cuanto antes la parte de la aplicación que más le interese).

Tras la primera reunión, se discutirá entre los integrantes del equipo de desarrollo distintos aspectos técnicos, de los cuales el más importante es decidir la aplicación que se va a usar como base. Previamente a la reunión con el cliente se deben haber estudiado distintas aplicaciones que se puedan usar para el proyecto de forma que se puedan decidir cuáles son las que más se acoplan a las necesidades del cliente, aunque normalmente se tenderá a tomar esta decisión según la experiencia previa de la empresa. Si no se encontrara una aplicación aprovechable se debería discutir aspectos como las distintas tecnologías a utilizar, el lenguaje de programación, etc.; aunque en este caso se presupone que hay una aplicación que se adapte al proyecto.

Con todas las decisiones tomadas se debe programar una nueva reunión con el cliente para decidir finalmente con qué departamento se va a empezar, y para definir los casos de uso (las distintas acciones que se pueden hacer en la aplicación) de dicho departamento. Dichos casos de uso se traducen en características de la aplicación y se decide cuales de ellas se van a implementar en la primera iteración. En esta reunión también se debe mostrar al cliente la aplicación base que se ha elegido, para que se haga una idea de lo que se puede y no se puede hacer. Este último paso es muy importante puesto que el cliente muchas veces no sabe lo que necesita con exactitud, y una demostración puede hacer que vea las cosas más claras.

Una vez concluidas las reuniones iniciales, se puede empezar la primera iteración, en la cual se implementaran las características que se decidieron en la última reunión. Para poder implementarlas, primero se debe pasar una etapa de aprendizaje de la parte del software existente requerido en la iteración. Para facilitar el trabajo posterior, lo primero que se debe hacer en esta etapa es desarrollar un mapa (mediante diagramas UML, ver **3.1**) de la aplicación de la que se parte. Para ello, es necesario utilizar herramientas de ingeniería inversa. A partir del mapa de la aplicación se podrá determinar qué tipo de cambios se pueden realizar, en cada caso, para conseguir los objetivos deseados. Evidentemente, este mapa sólo se tiene que elaborar en la primera iteración, en las siguientes ya estará disponible como punto de referencia.

Por otra parte, en esta etapa se debe comprobar si el software ya dispone de alguna de las características a implementar o si se puede aprovechar alguna de las que tiene para desarrollar las nuevas (reaprovechamiento de partes del código o modificación de alguna característica existente). También hay que estudiar la forma de integrar las características nuevas dentro de la aplicación base, ya que muchas veces puede ser necesario estudiar el código fuente de la aplicación original (para su posterior modificación), o ver qué mecanismos trae la aplicación para extenderla.

Tras el estudio del software, se deben modelar las características de forma independiente y teniendo en cuenta los resultados de dicho estudio. Los

cambios a realizar en la aplicación base se han de diseñar teniendo en cuenta que pueden ser de dos tipos:

- **Estructurales:** modifican o amplían la estructura original de la aplicación.

- **Funcionales:** añaden nueva funcionalidad.

La implementación de nuevas características normalmente se compondrá de un conjunto de cambios estructurales más un conjunto de cambios funcionales. Es importante tener en cuenta la complejidad que introducirá cada uno de los cambios diseñados y cómo afectará a la mantenibilidad de la aplicación.

Finalmente, se puede pasar a la implementación de las características nuevas. En la etapa de desarrollo e implementación es recomendable utilizar metodologías o técnicas de gestión de cambios (documentar los cambios y las posibles consecuencias futuras, por ejemplo), de forma que luego se pueda mantener una relación de los mismos para facilitar el mantenimiento futuro. Cada iteración debe durar una semana o, como mucho, dos, dependiendo de la disponibilidad del cliente para tener una reunión con el equipo.

Antes de dar por cerrada la iteración, el equipo de desarrollo se debe reunir para probar las nuevas características de la aplicación antes de mostrárselas al cliente. Con esto se pretende evitar que aparezcan errores durante la demostración o que, si los hay, tengan una justificación.

Tras la comprobación de las nuevas características, se debe programar una reunión con el cliente para mostrar los avances conseguidos durante la última iteración. En dichas reuniones, el cliente puede opinar sobre el funcionamiento de las características y pedir cambios sobre las mismas (puede que alguna acción les parezca incómoda de hacer tal y como se ha diseñado, por ejemplo). En la misma reunión, también se deben decidir las características que se implementarán en la siguiente iteración. Acabada la reunión, se sigue el ciclo con la siguiente iteración, que será igual que la anterior aunque realizando las correcciones requeridas por el cliente.

Este ciclo se repetirá hasta que se acabe de implementar todas las características de un departamento. En ese momento, se debe revisar todo lo hecho hasta el momento para localizar y solucionar el máximo número posible de errores; refactorizar el código, es decir, separar partes del mismo para que más adelante se puedan reutilizar; y, finalmente, documentar todo el trabajo realizado hasta el momento. Además de todo esto, en la siguiente reunión con el cliente se debe decidir cuál será el próximo departamento con el que se va a trabajar.

Para valorar esta metodología hay que ver si consigue solucionar algunos de los problemas planteados en el primer capítulo. La etapa de análisis queda resuelta con la reunión inicial con el cliente, en la cual se recogen todos los datos que nos puedan interesar de la empresa; y con la primera reunión del equipo de desarrollo, en la cual se analizan dichos datos y se toman las

decisiones convenientes. La recogida de requerimientos se hace en forma de nuevas características para la aplicación existente, de forma que son más claros que si se buscan unos requisitos más abstractos.

Por su parte, la etapa de aprendizaje, también se ve favorecida puesto que el conocimiento de la aplicación se va haciendo poco a poco, a medida que necesitamos conocer aspectos que permitan implementar las nuevas características. De esta forma el equipo de desarrollo va conociendo mejor la aplicación existente a medida que avanza el proyecto y, además, sólo se acaba conociendo aquellas partes de la aplicación que se necesitan, evitando incrementar la carga de esta etapa y, por lo tanto, su coste económico.

Las etapas de diseño y desarrollo, como el resto, se hacen por partes con lo cual es más fácil solventar errores que puedan aparecer. Además, en estas etapas se recomienda ir controlando los cambios realizados, de forma que se reduzcan los costes futuros de mantenimiento. En cuanto a la etapa de testeo, se soluciona con la reunión del equipo antes de la demostración con el cliente. Teniendo en cuenta que todo está orientado a las características, lo más adecuado es que las pruebas se hagan de forma manual, de forma que se pueda estimar si las características nuevas hacen lo que deberían desde el punto de vista de un usuario, por ello, parte del testeo se realiza en la propia demostración al cliente (el cual se deben validar los avances).

Para acabar, sólo queda comentar que esta valoración no será completa hasta que la metodología no se ponga en práctica con un proyecto real, ya que sobre el papel todo está muy bien, pero a la hora de ponerlo en práctica siempre surgen problemas inesperados. En el **capítulo 4** se puede ver la puesta en práctica de esta metodología, incluidos los problemas surgidos durante el desarrollo.

CAPÍTULO 3. HERRAMIENTAS DE DESARROLLO

Cualquier proyecto de ingeniería de software requiere hacer uso de distintas herramientas. Se pueden encontrar multitud de herramientas, y es muy importante elegir las más adecuadas para cada caso particular. En este capítulo se exponen distintas herramientas que nos pueden ser útiles en cada etapa del desarrollo de un proyecto de ingeniería de software.

Se ha dividido el capítulo en distintos apartados, separando las distintas herramientas según su finalidad en el proceso de desarrollo. Al final del capítulo se incluye una explicación de las herramientas elegidas de cara al caso real.

3.1 Herramientas de diseño

Este tipo de herramientas, como su nombre indica, nacen con el objetivo de facilitar y dar unas pautas a la hora de diseñar piezas de software. En esta línea, se proponen distintos esquemas o tablas para documentar y planificar la estructura de la aplicación a desarrollar. Actualmente, la principal herramienta de diseño es UML [2], por lo cual este apartado está dedicado de forma exclusiva a este lenguaje y a aplicaciones que nos permiten trabajar con él.

De forma resumida, UML (siglas inglesas de *Unified Modeling Language*, o lenguaje unificado de modelado) proporciona mecanismos estándar para visualizar, especificar, diseñar y documentar sistemas de software. En términos generales, UML nos permite dibujar el “plano” de una aplicación, es decir, su modelo, incluyendo aspectos generales tales como procesos de la lógica de negocio, funciones, componentes, o incluso otros más concretos como podrían ser diseños de bases de datos o expresiones propias de un lenguaje en particular.

Principalmente, UML propone distintos tipos de diagrama que pretenden plasmar cada aspecto de la aplicación que se está diseñando. En concreto existen 11 tipos de diagrama distintos. En el **anexo A** se puede encontrar un listado de los tipos de diagrama, con la descripción y un pequeño ejemplo de cada uno.

Por otro lado, han surgido una serie de aplicaciones que pretenden facilitar todo el proceso de diseño mediante UML: los entornos de diseño. Este tipo de aplicaciones permiten crear y editar esquemas UML de forma muy sencilla y cómoda. Habitualmente, se componen de un editor gráfico (similar a cualquier programa de dibujo o edición fotográfica) que incluye distintas formas que se corresponden con los distintos objetos que se pueden encontrar en los diagramas UML. Aparte de esto, también suelen incorporar herramientas adicionales, como por ejemplo generadores automáticos de código (los cuales

generan el esqueleto del código según nuestro diseño), ingeniería inversa (para dibujar diseños a partir del código de alguna aplicación), etc.

En el presente proyecto se usará el editor UML de la suite *Visual Paradigm* [4]. Este editor es muy completo y dispone de múltiples características, entre ellas lo más destacable es la generación automática de código, el hecho de que permite realizar todos los diagramas UML existentes en la especificación 2.0 y la posibilidad de usarlo conjuntamente con un servidor de trabajo en equipo (el *Teamwork server*), que funciona de forma similar a un repositorio CVS. En este caso no se ha probado otros editores, el motivo de esto es que la empresa desarrolladora tiene comprada la licencia de esta suite de aplicaciones y no tendría sentido comprar una nueva licencia para otro software (teniendo en cuenta que es un software muy potente).

Para acabar, en la siguiente imagen se puede ver la interfaz del editor de UML *Visual Paradigm*.

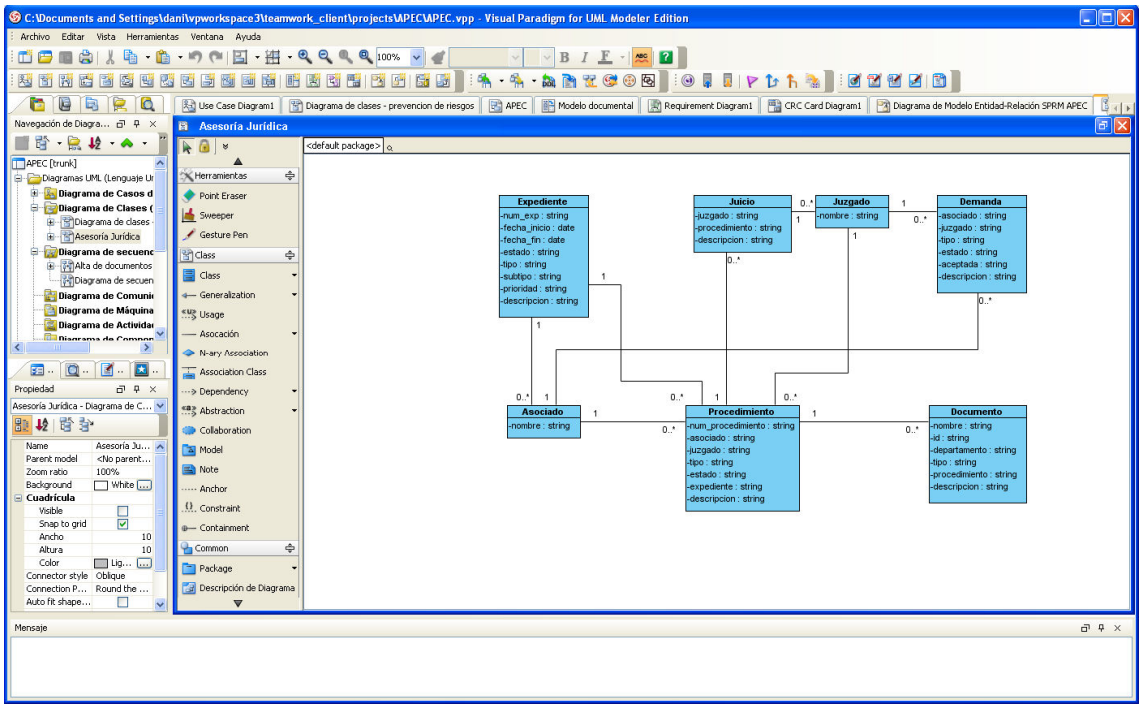

Fig. 3.1 Interfaz gráfica de *Visual paradigm*

3.2 Herramientas de programación

Este tipo de herramientas tienen como finalidad facilitar a los desarrolladores a escribir y testear el código fuente de que se compone la aplicación. Entre las herramientas de este tipo se incluyen entornos de desarrollo (IDE), compiladores o herramientas de mecanización de procesos, como pueden ser *"Ant"* [5] o el más reciente *"maven"* [6]. En este apartado se describe la función

de cada tipo de herramienta de programación, además de comentar y comparar distintos ejemplos.

3.2.1 IDE

Las siglas inglesas IDE significan *Integrated Development Eviroment*, es decir, entorno de desarrollo integrado. Este tipo de aplicaciones nos proporcionan ciertas facilidades a la hora de escribir código. Cada IDE tiene características distintas, pero las más comunes son el coloreado del código (con la finalidad de identificar visualmente distintas partes del código como, por ejemplo, variables, comentarios, funciones, etc.), integración con compiladores para compilar el código con un sólo clic y función de *debugger* para facilitar la localización de errores de programación.

Algunos de los IDE's más famosos son: *Microsoft Visual Studio* [7] (pensado para C, C++ y C#), *Zend* [8] (para editar código PHP), *netbeans* [9] (Java) o *eclipse* [10] (en principio pensado para Java, pero dispone de plugins para funcionar en multitud de lenguajes de programación).

La elección de un IDE u otro se ha de llevar a cabo primero teniendo en cuenta el lenguaje de programación que vamos a usar y luego las características que cada uno nos ofrece. Otro dato interesante es el precio de cada uno, sobre todo cuando el presupuesto de la empresa no es excesivamente elevado.

3.2.2 Compiladores

En cuanto a compiladores, hay que destacar que cada lenguaje compilado tiene el suyo y poco más hay que decir, simplemente son programas, habitualmente en modo texto, que permiten convertir el código a un lenguaje entendible para la máquina. Por ello, en este apartado, más que explicar los compiladores como herramienta, se describen herramientas que nos facilitan las tareas "mecánicas" (en el sentido de que son siempre iguales) de compilación, empaquetado y despliegue de las aplicaciones. Las herramientas de este tipo más famosas y utilizadas son *Ant* y *Maven*. Las dos aplicaciones están desarrolladas por la *Apache Software Foundation* [11], y se puede decir que la segunda es una evolución de la primera.

3.2.2.1 Ant

Ant es una herramienta de construcción de software. Nació con el objetivo de superar las limitaciones que tenían las herramientas existentes con anterioridad (*make*, *gnumake*, *nmake*, *jam*, etc.). La ventaja que introduce respecto a estas es que no depende de las órdenes de la línea de comandos de cada sistema operativo, sino que está basado en archivos de configuración XML y en clases java. Para conseguirlo, *ant* incluye multitud de funciones que sustituyen a los

comandos clásicos de una línea de comandos. Incluye funciones de creación/eliminación de archivos y carpetas, funciones de empaquetamiento de software, etc. Por todo ello, teniendo en cuenta que tanto java como XML son multiplataforma, se presenta como una herramienta idónea para soluciones independientes del sistema operativo.

Por otro lado, *ant* también comporta una serie de limitaciones. La primera de ellas es que es una herramienta basada en XML, con toda la complicación que ello supone para nuevos usuarios y en proyectos de gran envergadura, en los cuales los archivos *ant* se hacen enormes y muy complejos. Otra limitación seria sus reglas de manejo de errores y el hecho de que no tiene persistencia de estado, con lo cual no puede ser usado con confianza para manejar construcciones largas (de uno o más días).

3.2.2.2 Maven

Maven es una herramienta muy similar a *ant*, disponiendo de una funcionalidad prácticamente idéntica, pero introduce una mejora muy importante: resolución automática de las dependencias. En el fichero de configuración XML (llamado project.xml en la primera versión de la herramienta y pom.xml en la segunda), además de las tareas automáticas que se quieren ejecutar (como en el caso de *ant*, compilar el proyecto, ejecutar pruebas unitarias, empaquetar, etc.), se deben definir las dependencias. Cuando se ejecute el archivo de configuración *Maven*, se comprobarán las dependencias de forma automática y, en caso de que faltase o existiese una nueva versión de alguna, se descargaría de un servidor repositorio. Otra ventaja que ofrece frente a *ant* es la generación automática de informes de construcción, tales como *javadoc*, métricas, resultados de las pruebas unitarias, etc.

Por otra parte, *Maven* presenta también una desventaja principal: la rigidez. Los creadores de esta herramienta han querido fomentar buenas costumbres de desarrollo obligando a los proyectos a tener siempre la misma estructura, creado una forma estándar de construir las aplicaciones. Esto lo que consigue es reducir la flexibilidad de la herramienta. Ahora bien, esto, a su vez, también supone una ventaja (según se mire), ya que la herramienta proporciona una estructura fija para todos los proyectos, y el archivo de configuración (pom.xml) será siempre prácticamente igual porque no hay que definir el proceso de construcción, con lo cual también se ahorra trabajo de configuración.

3.3 Herramientas de gestión de la documentación

Como su nombre indica, este tipo de herramientas nos ayuda a organizar la documentación generada, de forma que sea accesible para todos los miembros del equipo. En este apartado describiremos las funciones, ventajas y desventajas de tres gestores documentales, cada uno de ellos con un enfoque muy distinto. Los tres programas analizados son: *dokuWiki* [12], *Knowledge*

tree [13] y *Alfresco* [14], elegidos por ser los más conocidos y usados en la comunidad de internet.

3.3.1 DokuWiki

Desde el nacimiento de la *wikipedia*, el concepto de "*Wiki*" está en boca de todos. Se denomina "*Wiki*" a un tipo de páginas web en las que los usuarios de la misma pueden editar o añadir contenido a través del navegador Web. Otra característica principal es que los títulos de las páginas de un *wiki* deben ser únicos, de forma que se pueda generar, de forma directa, enlaces de unas páginas del *wiki* a otras, para, de esta forma, mantener una relación entre los temas que se pueden encontrar en el *wiki*. Con todo ello, lo que al final se obtiene es un gestor documental, ya que permite subir (en forma de páginas del *wiki*) documentos al servidor y mantener un orden y una relación entre ellos. La aplicación *wiki* más usada (además de ser la que alberga a la propia *wikipedia*) es *mediaWiki* [15], pero en este caso se ha preferido analizar *DokuWiki*, ya que está pensado para la documentación de proyectos en pequeñas y medianas empresas.

Principalmente, lo que lo diferencia de *mediaWiki* es que no es necesario el uso de bases de datos para su funcionamiento, puesto que la información generada se almacena en archivos de texto planos. Esto hace que simplemente disponiendo de un servidor Web con intérprete de PHP instalado, se puede tener instalada la aplicación en pocos segundos (lo que tarde en descomprimirse el archivo ZIP en el cual viene empaquetada). Por lo demás funciona de forma muy similar a *mediaWiki*: la sintaxis para escribir los documentos es la misma; dispone de control de acceso para decidir quién puede ver/editar cada página; soporte para archivos multimedia, etc. La siguiente figura muestra la interfaz web de la aplicación.

Fig. 3.2 Interfaz web de *DokuWiki*

Resumiendo, lo que nos ofrece esta aplicación es facilidad y rapidez de configuración. Por otro lado, el punto negativo es que no nos ofrece funciones avanzadas como flujos de trabajo o definición de distintos tipos de documento. De todas formas, se trata de una buena opción si lo único que queremos hacer es documentar la aplicación (manuales de funcionamiento, instalación o definición de características) de forma compartida.

3.3.2 Knowledge Tree

Esta aplicación es una de las más famosas en cuanto a gestión documental se refiere. Está destinado a todo tipo de empresas y su principal punto fuerte es que ofrece un buen conjunto de funcionalidades avanzadas y, sobre todo, una gran simplicidad a la hora de configurarlo y personalizarlo. Su interfaz gráfica es muy intuitiva y no es nada complicado crear carpetas, subir documentos o, incluso, crear flujos de trabajo avanzados. También desde la interfaz gráfica nos permite crear nuevos tipos de documento y los metadatos asociados a los mismos, así como editar los tipos de documento incluidos por defecto. En la siguiente figura se puede ver dicha interfaz.

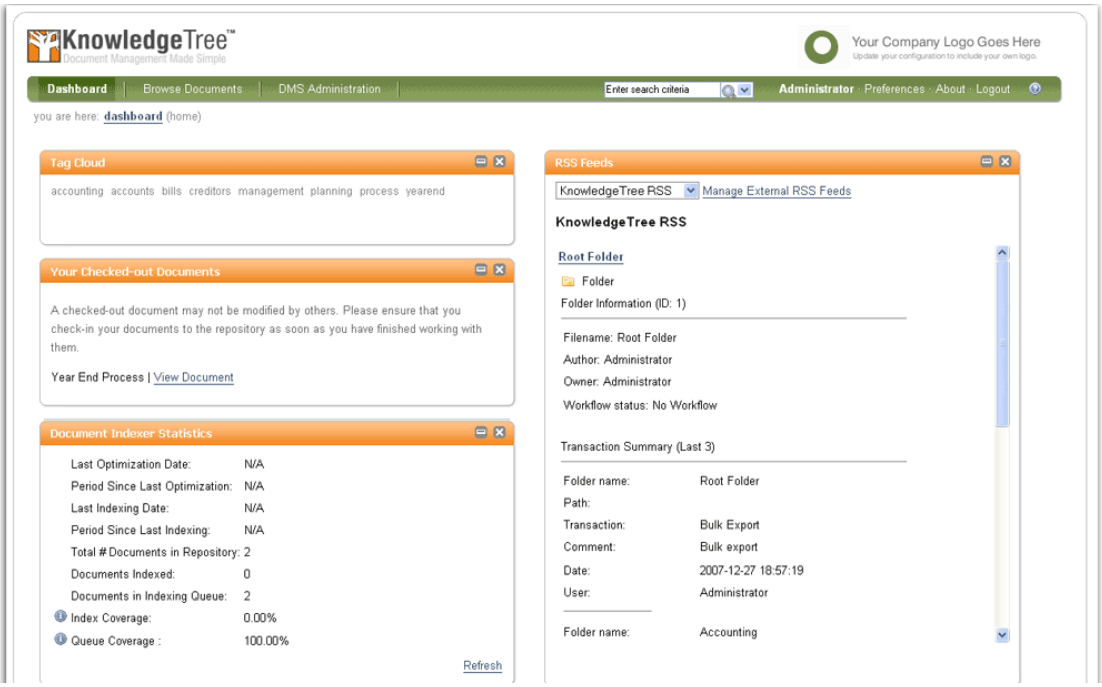

Fig. 3.3 Interfaz gráfica de *knowledge tree*

Como punto flojo podemos decir que es complicado añadir nuevas funcionalidades, puesto que no dispone de ningún mecanismo de instalación de nuevos módulos y extensiones, con lo cual si queremos añadir nuevas

características a la aplicación tenemos que modificar directamente el código fuente, con todos los riesgos que ello conlleva.

Como valoración general podemos destacar, por encima de todo, su gran simplicidad. Se podría implantar en una empresa mediana en poco rato y que los usuarios lo empezaran a utilizar puesto que su interfaz no necesita explicaciones. También destacar el amplio abanico de características, que hacen que, en muchos casos, no sea necesario retocar el código para adaptarlo a nuestras necesidades.

3.3.3 Alfresco

Esta es, sin duda, la aplicación más compleja de las tres comentadas, aunque, a su vez, también es la más potente y la que más características ofrece. Aun con lo dicho, trabajar con la interfaz gráfica de esta aplicación es muy simple (tanto como *Knowledge Tree*). Dicha interfaz es similar al gestor de archivos que podemos encontrar en cualquier sistema operativo y las acciones de subir archivos y crear carpetas es muy simple. La complejidad de esta aplicación se encuentra a la hora de personalizarla, ya que todas las personalizaciones se tienen que hacer “a mano”, es decir, mediante archivos XML o directamente mediante programación. El punto fuerte de que la personalización se haga de este modo es que permite una personalización total, hasta el punto de que se le pueden llegar a añadir nuevas funcionalidades o facilitar la integración con otras aplicaciones.

Entre sus características principales se encuentra la posibilidad de acceder al contenido gestionado por la aplicación desde diferentes vías: FTP, CIFS (carpetas compartidas), por la propia interfaz Web, mediante enlaces de descarga directa y SOAP. Otra característica interesante son las reglas de contenido, mediante las cuales podemos efectuar diferentes acciones (mover a una carpeta determinada, ejecutar un script, añadir un aspecto, etc.) sobre el contenido entrante, saliente o modificado. La siguiente imagen muestra la interfaz gráfica de la aplicación.

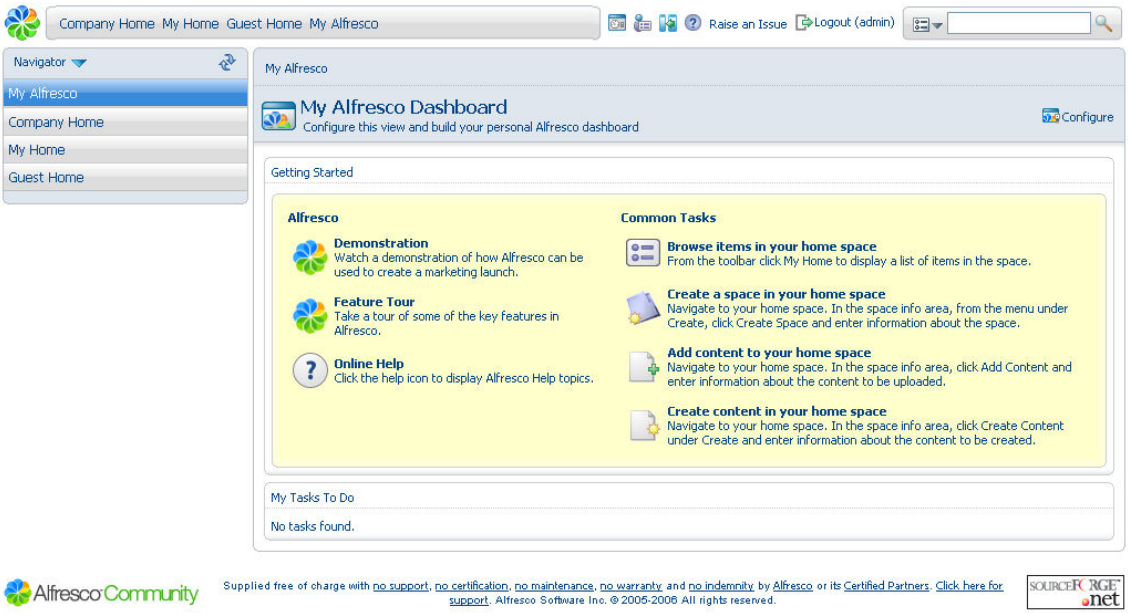

Fig. 3.4 Página de inicio de *Alfresco*

El punto negativo se encuentra en la relativa dificultad de instalación y configuración. Si no se usa la versión que viene con todo incluido (Servidor de aplicaciones, base de datos y el propio *Alfresco*), es muy difícil que la primera vez que se despliega la aplicación funcione correctamente, ya que hay muchos detalles que configurar, dependiendo del servidor de aplicaciones utilizado, y es fácil obviar algún paso. De todas formas, la aplicación está muy bien documentada y se puede encontrar solución a todos los problemas que puedan surgir.

Como valoración general, se puede decir que es una aplicación muy potente, pero a su vez muy compleja, lo cual dificulta su entendimiento. Es recomendable si se desea aplicar flujos de trabajo complejos al contenido o se requiere un orden exhaustivo en la información almacenada. También es muy útil cuando es necesario un gestor documental fácilmente integrable con otras aplicaciones (gracias a sus múltiples vías de acceso a los datos almacenados).

3.4 Herramientas de gestión de proyecto

Este tipo de herramientas ayudan al director del proyecto a planificar los distintos eventos del proyecto, así como asignarlos a los distintos componentes del equipo de desarrollo. Los eventos pueden ser desde la definición de tareas hasta la programación de reuniones o plazos de entrega.

3.4.1 DotProject

DotProject [16] es una completa aplicación web de gestión de proyectos, diseñada para proporcionar funciones de control y esquematización de proyectos de cualquier tipo. En esta línea, su principal función es estructurar una serie de tareas y su planificación (plazos, fechas, etc.), dentro del marco de un proyecto previamente definido. Por otro lado, aporta una serie de funcionalidades extra para ayudar a la gestión de los proyectos, como pueden ser diagramas de Gantt automáticos, calendario con la planificación de tareas, alertas por e-mail (para asignaciones de tareas, actualizaciones, etc.), estructuración de tareas por compañías (las cuales se pueden organizar por tipo de empresa), repositorio de archivos o foros de comunicación entre desarrolladores.

El punto negativo de esta aplicación se puede encontrar en la complejidad inicial de la interfaz gráfica. La primera vez que se entra en *dotProject* es algo complicado aprender cómo funciona todo, el problema es que los menús son poco amigables, con muchas pestañas que hay que explorar con calma para aprender su funcionamiento, con lo cual puede echar para atrás en un principio. Estos problemas acaban una vez se ha examinado del todo la aplicación, a partir de ese momento no es difícil de usar puesto que todo se hace más o menos de la misma forma.

Otro punto negativo de la interfaz gráfica es que tiene una estética algo anticuada, pero eso no es un problema mayor siempre y cuando la aplicación sea potente en cuanto a características, y *dotProject* es sobresaliente en funcionalidad. En la siguiente figura se puede ver una captura de la interfaz gráfica de *dotProject*.

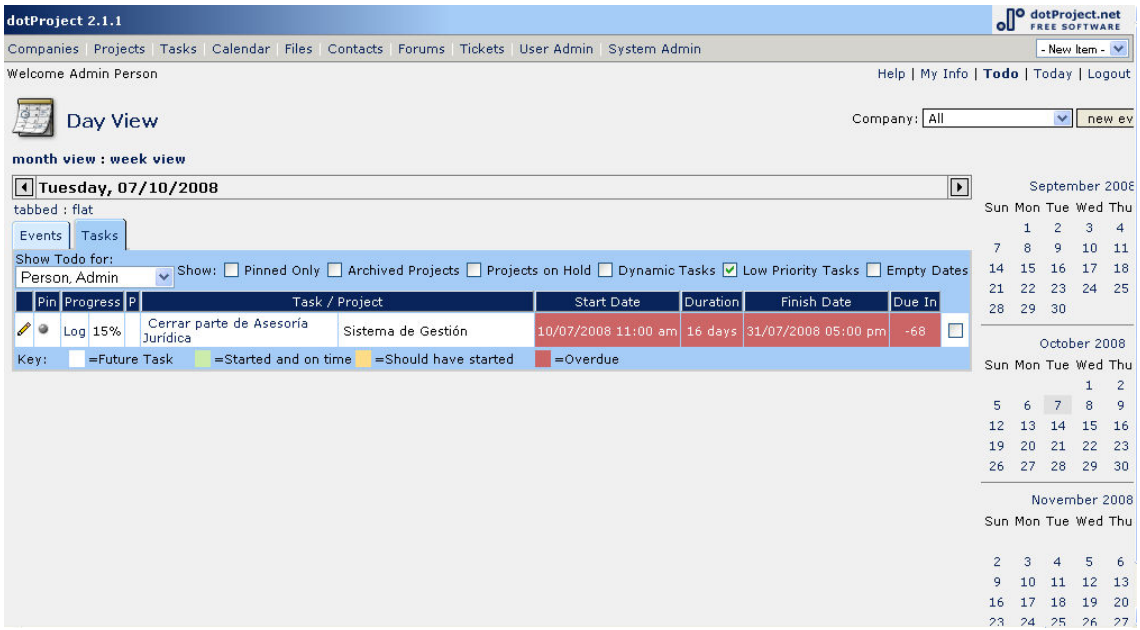

Fig. 3.5 Interfaz gráfica de *dotProject*

3.4.2 Project Open

Project open [17] sigue un concepto similar al de *dotProject*, es decir, se trata de una aplicación web de gestión de proyectos. El punto fuerte de esta aplicación es la funcionalidad, mucho más completa que la de *dotProject*, sobre todo en el área de finanzas. En este sentido *dotProject* proponía muy pocas opciones (costes de cada tarea y cuenta total de un proyecto), en cambio, con *Project Open* podemos calcular impuestos, definir el salario de cada trabajador, hacer desgloses de costes, marcar pagos como facturados o recibidos, etc. También propone un control más exhaustivo de los empleados, de los cuales se puede marcar cuando están ausentes (por baja o vacaciones, por ejemplo), las horas que han estado trabajando en una tarea (calculándose de forma automática el coste de esas horas), etc.

De la misma forma, se debe destacar como aspecto positivo el hecho de que disponga de una interfaz gráfica más acorde con los tiempos que corren y, sobre todo, mucho más clara, tanto en su presentación como en su facilidad de uso. En este aspecto también está mejor que su rival en esta comparativa. Otro punto destacable es su facilidad de instalación, ya que se distribuye en forma de imagen de *VMWare* [18] con todo lo necesario para que funcione, tan sólo hay que ejecutar la imagen en el *VMWare Player* (aplicación gratuita) y en pocos minutos se puede tener el servidor funcionando. El punto negativo de este tipo de distribución es que es bastante pesada (1,4 GB). La siguiente imagen muestra la interfaz gráfica de la aplicación.

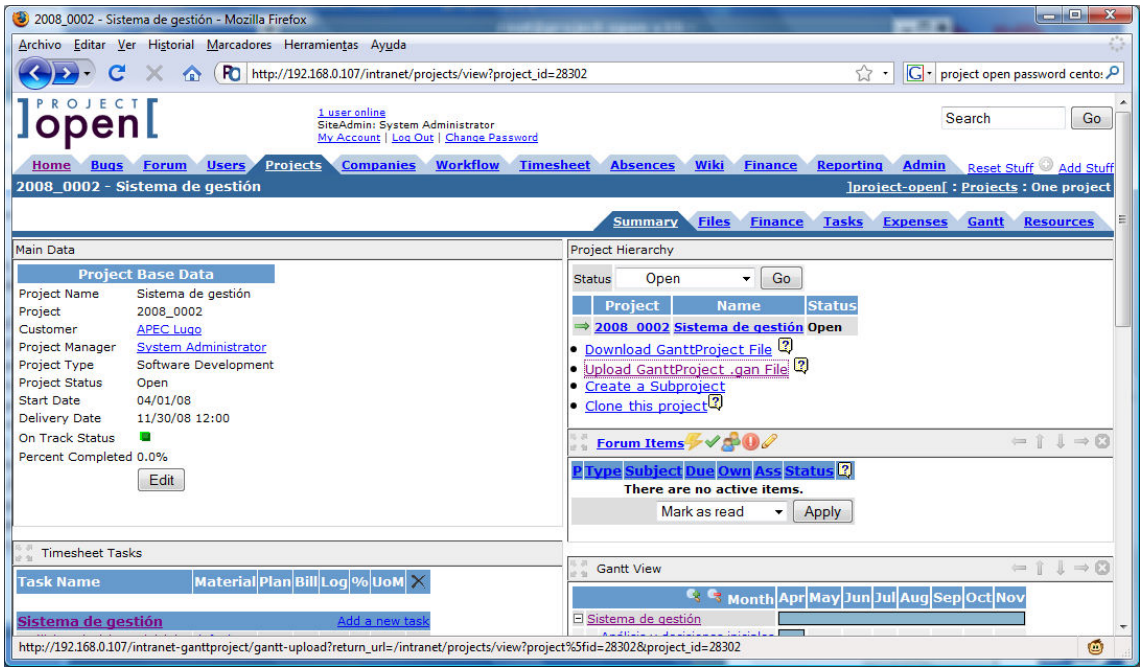

Fig. 3.6 Interfaz gráfica de open project

Resumiendo podemos decir que se trata de una herramienta muy potente, que puede hacer lo mismo que su rival y más cosas, y todo ello con un aspecto mucho más moderno y fácil de usar.

3.4.3 Microsoft Project

En este caso el concepto de aplicación es muy distinto al de los dos anteriores. *Microsoft Project* [19] es una aplicación de escritorio de la suite de ofimática *Microsoft Office*. En ella principalmente se nos permite hacer la planificación de un proyecto (definición de tareas, asignación de recursos, plazos, etc.), para luego mostrarla con distintas vistas (diagramas de Gantt, de recursos, de relaciones, etc.). Es decir, que más que un programa de gestión de proyectos es un programa de planificación, ya que no implementa flujos de trabajo ni demasiadas opciones a nivel financiero, simplemente permite definir diagramas de planificación. En la siguiente figura se puede ver el aspecto de Microsoft Project.

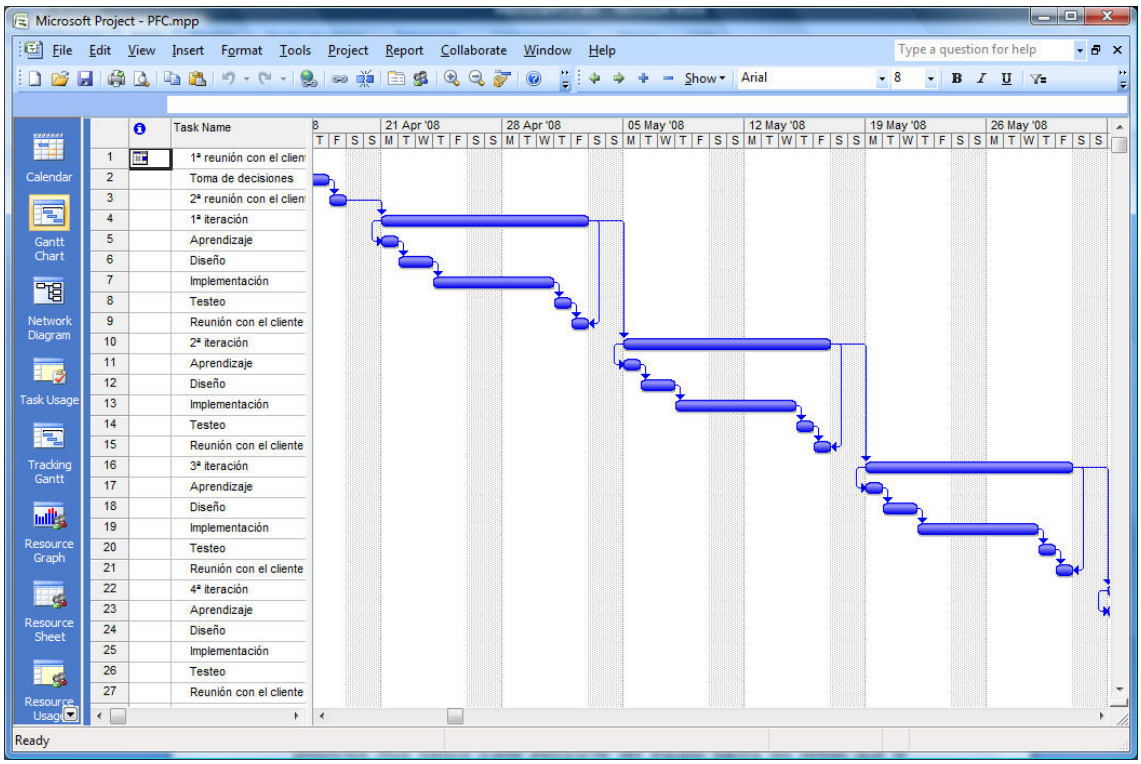

Fig. 3.7 interfaz gráfica de Microsoft Project

Para compartir las planificaciones creadas con los integrantes del equipo de desarrollo, es necesario hacer uso del servidor *Microsoft Project server*. De esta forma se podría usar esta aplicación como gestor de proyectos compartido (compartición de proyectos, diagramas e información asociada a los mismos), al poder ir actualizando los diagramas a mano sabiendo que todo el equipo va a poder ver la nueva información.

Este concepto de aplicación tiene una ventaja clara sobre el expuesto en las dos aplicaciones anteriores. Esta ventaja no es otra que la de poder acceder a la aplicación sin necesidad de tener conexión a internet, algo muy útil si se tiene que ir a realizar tareas fuera de la empresa. Y si se hiciera algún cambio sin tener conectividad con el servidor de trabajo en equipo se puede actualizar más tarde al volver a un lugar con conectividad.

El principal punto negativo de esta aplicación es que es de Microsoft, lo cual quiere decir que su precio es bastante elevado. Además las licencias del cliente y el servidor van por separado así que, si se quiere un entorno de gestión para trabajar en equipo, la cuenta aun sube más.

3.4.4 Open Workbench

Esta aplicación [20] se presenta como una alternativa gratuita a *Microsoft Project*. Principalmente hace lo mismo que *Microsoft Project*, con la única diferencia de que tiene algún que otro tipo de diagrama menos, pero los más importantes están. Por lo tanto, es una opción mucho más rentable, aunque no totalmente gratuita porque el servidor para compartir los diagramas disponible para esta aplicación sí que es de pago, por lo que si queremos beneficiarnos de sus posibilidades sí que habrá que realizar un desembolso económico.

Como se puede ver en la siguiente figura, la interfaz gráfica de la aplicación es muy similar (aunque un poco más pobre de aspecto) a la de *Microsoft Project*.

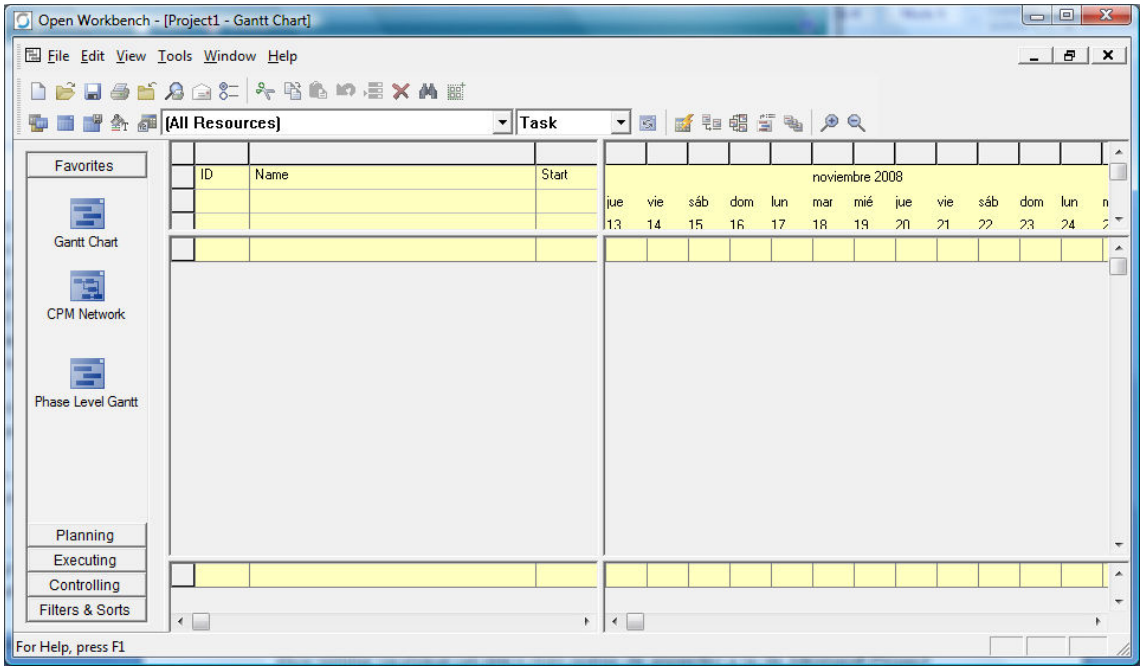

Fig. 3.8 interfaz gráfica de Open workbench

3.5 Herramientas de trabajo en equipo

Las herramientas de trabajo en equipo permiten compartir los avances en el proyecto entre los distintos grupos de trabajo que componen el equipo de desarrollo. Las herramientas descritas en los siguientes subapartados (CVS [21] y SVN [22]), nos permiten almacenar todo el código fuente desarrollado en un servidor para que todos los componentes del equipo de desarrollo puedan conectarse a este servidor y disponer de todas las partes del proyecto. Estas herramientas son especialmente útiles cuando el trabajo de un componente del equipo depende del de otro.

3.5.1 CVS

Las siglas CVS responden a *Concurrent Versions Systems*, es decir, sistema de versiones concurrentes, lo que quiere decir que se trata de una aplicación de control de versiones. Su funcionamiento está basado en una arquitectura cliente-servidor, en la cual el servidor se encarga de gestionar las versiones de un determinado código fuente, mientras que los clientes tienen dos funciones: la primera es descargarse el código actual o versiones antiguas del mismo para hacer comparaciones de versiones; la segunda es subir los últimos cambios realizados al código.

El código descargado del servidor se modifica en el sistema local, y luego se suben los cambios al servidor, el cual almacena un historial de los distintos cambios realizados y guarda las modificaciones con el último número de versión, de forma que si dos usuarios están trabajando con la misma versión y hacen cambios más o menos a la vez, el servidor le notificará, al cliente que actualice más tarde, que antes ha actualizado alguien y podrá comparar los cambios para aceptarlos y actualizar su versión local o rechazarlos e imponer sus propias modificaciones.

En sus inicios, CVS estaba pensado para funcionar en sistemas *unix*, pero, en la actualidad, existen versiones de CVS (tanto clientes como servidores) para los sistemas operativos más comunes, incluido Windows. Esto facilita que los miembros de un mismo grupo de trabajo puedan estar trabajando con distintos sistemas operativos sin crear conflictos.

En general es una aplicación muy útil y fácil de usar, pero existen ciertas limitaciones en el protocolo que hacen que en ocasiones sea muy tedioso trabajar con él. Un ejemplo de estas limitaciones seria la imposibilidad de renombrar los archivos, tarea para lo cual hay que eliminar el archivo original y luego subirlo ya renombrado. Resumiendo, es muy útil pero mejorable, de ahí el desarrollo de la aplicación de la que hablaremos en el siguiente apartado: SVN.

3.5.2 SVN

En este caso SVN no responde a unas siglas determinadas, sino que se trata del nombre de la herramienta por línea de comandos que sirve de cliente de la aplicación. Su nombre completo es *Subversion*, y nació con la finalidad de

sustituir a CVS y solventar sus limitaciones de diseño. Principalmente funciona de forma similar a CVS, pero introduce una serie de mejoras:

- Se lleva un historial de los archivos y directorios a través de copias y renombrados.

- Se envían sólo las diferencias entre los archivos del servidor y el cliente (en CVS se envían archivos completos).

- Maneja de forma eficiente los archivos binarios (CVS los trata como archivos de texto).

- Permite el bloqueo de archivos.

- Puede ser servido mediante apache, a través de WebDAV.

- Al usarse integrado a apache, puede utilizar todas las opciones que este provee (BBDD, LDAP, PAM, etc.) a la hora de autenticar el acceso a los archivos.

Estas mejoras hacen de SVN un protocolo más completo a la hora de versionar archivos, y todo manteniendo la misma simplicidad de CVS de cara al usuario. Por otro lado, no se puede decir que SVN sea perfecto ni mucho menos, puesto que aun incluye una serie de debilidades que lo hacen mejorable. De entre estas debilidades destacan el hecho de que el renombrado de archivos no sea 100% completo (puesto que se trata de una operación de copia más otra de borrado); y que sigue dificultando el control de los cambios realizados

3.6 Herramientas de control de calidad

Más que un tipo de herramientas, en este apartado se define lo que se conoce como métricas en ingeniería de software. A grandes rasgos las métricas son una serie de pautas objetivas que nos permitirán analizar la calidad del sistema en distintas etapas de su desarrollo. De forma más concreta, las métricas son medidas cuantitativas de algunas propiedades o características de una pieza de software.

Existen multitud de medidas distintas que se pueden tomar sobre una pieza de software. En las siguientes líneas describiremos aquellas consideradas de uso más común:

- Número de líneas de código.
- Complejidad ciclomática.
- Bugs por línea de código.
- Cobertura del código.
- Métrica de punto función.
- Número de líneas de requerimientos.
- Número de clases e interfaces.

- Cohesión.
- Dependencia.

Con estas medidas, lo que se pretende conseguir es determinar el tamaño del software, su complejidad, el coste de producción aproximado, el tiempo necesario para el desarrollo y, finalmente, la calidad del código. De esto se deduce (teniendo en cuenta que necesitamos hacer estimaciones iniciales acerca del desarrollo) que algunas medidas se toman en la etapa de diseño y otras una vez se ha puesto en funcionamiento la aplicación o alguna parte de ella.

Para realizar el cálculo de las métricas existen numerosas herramientas. Algunas de ellas sólo se encargan de hacer una de las mediciones, mientras que otras se encargan de analizar distintos aspectos del software. También se pueden encontrar herramientas que lo que analizan no es código, sino diagramas UML, con lo cual nos permiten analizar el proyecto en la etapa de diseño, cuando aun no hay código que analizar. En la siguiente lista se pueden ver algunos ejemplos de herramientas de cálculo de métricas.

- **SDMetrics:** analiza la estructura de los diseños UML y detecta posibles errores de diseño. Predice aspectos como la mantenibilidad o la complejidad de la aplicación.

- ***Semantic Designs*:** se trata de una serie de herramientas que se centran en el análisis del código fuente. Entre ellas se encuentran varias dedicadas al cálculo de métricas en distintos lenguajes de programación (c#, Java, VBScript y Logix500). Entre todas, pueden realizar prácticamente todas las métricas existentes relacionadas con el código fuente.

- ***JHawk*:** esta es una herramienta de cálculo de métricas para Java. Dispone de un buen número de métricas estándar, además de algunas diseñadas por los creadores de la aplicación.

- ***Project analyzer*:** esta herramienta está pensada para analizar código Visual Basic, VB.NET y VBA. Entre sus numerosas funciones incluye el cálculo de hasta 180 tipos de métricas diferentes.

- ***Sloccount*:** única herramienta que se ha encontrado que funcione con código PHP, lamentablemente sólo contiene una métrica: número de líneas de código.

De todas estas herramientas, la única que puede ser de utilidad (en el caso real que se va a tratar) es *Sloccount*, porque es la única que trabaja con código PHP. Esto significa que, sobre el código, sólo se podrá calcular la métrica del número de líneas, lo cual no proporciona demasiada información (el programa hace una estimación del tiempo que se tardaría en desarrollar todo el código y de los programadores que se necesitarían, pero todo eso te lo dice una vez el código ya está hecho, con lo cual la utilidad es mínima).

3.7 Selección de las herramientas a usar en el caso real

Tras probar y evaluar todas las herramientas descritas, el siguiente paso es decidir cuáles de estas se usarán en el caso real. En las siguientes líneas se puede ver las decisiones tomadas, así como el motivo por el cual se ha tomado cada una de ellas.

Para empezar, hay que comentar la decisión tomada con la herramienta de diseño. En este caso sólo se dispone de una herramienta, UML, por lo tanto la decisión se limitará a qué tipo de diagramas se van a usar. En este caso, y puesto que las metodologías ágiles piden más tiempo de implementación y menos de "papeleo", sólo se usarán tres tipos de diagrama: el diagrama de casos de uso, el de clases, y el de secuencia. Con el primero se consigue tener una visión gráfica de los distintos procedimientos a implementar; el segundo permite ver las piezas, en forma de clases, que compondrán la solución final; y con el tercero se consigue definir los pasos a seguir en cada caso de uso.

La elección del entorno de programación se ha basado principalmente en el lenguaje de programación. Puesto que la aplicación de la que se parte está desarrollada en PHP, no queda más remedio que usar una herramienta pensada para este lenguaje, de las cuales la mejor es *Zend*. Por otra parte, en el proyecto también participan algunas aplicaciones desarrolladas en Java. En el caso que se tuviera que editar parte del código de estas últimas, se usaría *Eclipse*. En cuanto a las herramientas de compilación (*ant* y *maven*), sólo se usarán en el caso de que haya que editar código java, caso en el cual seguramente se usará *ant* por dos motivos principales: el primero es que dicha herramienta viene incluida con el entorno de desarrollo *eclipse*; el segundo es que si hay algo que editar será pequeño con lo cual el archivo de configuración de *ant* no será muy complejo (como mucho compilar y empaquetar), con lo cual no se aprovechan las ventajas que ofrece *maven*.

En el caso de los gestores documentales, se ha elegido *Alfresco*, el cual, aun siendo el más complejo de los tres presentados, es el más potente y la dificultad que introduce no es tanta si se tiene en cuenta que el equipo de desarrollo ya tenía experiencia previa con esta aplicación. De hecho la empresa desarrolladora ya disponía, previamente al presente proyecto, de un servidor *Alfresco* en el cual almacena la documentación acerca de los proyectos y trabajos que va realizando.

De las herramientas de gestión de proyectos, sólo se usará *Microsoft Project* u *Open workbench* para la planificación del proyecto mediante un diagrama de Gantt. Si bien es cierto que, de esta forma, no se consigue una gestión completa del proyecto, pero al ser un proyecto relativamente pequeño no se estima necesaria una gestión a más alto nivel.

En cuanto a las herramientas de trabajo en equipo, si se tuviera que usar alguna se usaría SVN, por razones evidentes (es prácticamente igual que CVS pero con algunas mejoras). Por otro lado, en este proyecto los cambios se

realizarán directamente en un servidor de pruebas para luego pasarlo (una vez validadas las nuevas características) a un servidor “estable”, por lo tanto, el código se hace accesible (entrando en los servidores) a cualquier miembro del equipo. Por todo esto, no se usará ninguna de estas herramientas.

Finalmente, sólo falta por comentar las herramientas de control de calidad. Como ya se ha dicho en el apartado relacionado con las mismas, de las herramientas encontradas de cálculo de métricas, sólo se ha encontrado una que permita trabajar con código PHP, y sus funciones son mínimas, con lo cual al acabar el desarrollo se calculará el número de líneas de código, pero más de forma anecdótica que por la información que esto pueda proporcionar.

CAPÍTULO 4. CASO REAL

En este capítulo se describe todo lo relacionado con el proyecto real en el cual se está trabajando actualmente. Puesto que la temática del proyecto es acerca de metodologías de desarrollo, se pondrá especial énfasis en el seguimiento de la metodología descrita en el apartado **2.3**. Esto no significa que se vaya a descuidar el producto desarrollado en sí mismo, pero no se le da tanta importancia porque la finalidad del presente proyecto es estudiar y evaluar que metodología es mejor para un tipo de proyecto en concreto, no llevar un proyecto de desarrollo a buen puerto.

El capítulo se divide en tres partes principales. Los primeros apartados se encargan de describir el proyecto, sus requisitos, las decisiones iniciales, etc. La segunda parte describe la aplicación base, en este caso SugarCRM [23]. Finalmente, la tercera parte se compone de una serie de apartados que describen los distintos aspectos del desarrollo del proyecto.

4.1 El proyecto

En este apartado y sus subapartados se describen cuestiones generales acerca del proyecto en curso. Primero se incluye una descripción del mismo, después se definen los requisitos generales y, finalmente, se exponen las decisiones tomadas en base a las necesidades del proyecto.

4.1.1 Descripción

El cliente es una asociación provincial de empresarios de la construcción. En los últimos años ha crecido de forma exponencial su número de asociados, lo cual reporta un volumen de datos muy difícil de gestionar sin las herramientas adecuadas. Actualmente disponen de una aplicación que les permite gestionar los servicios ofrecidos a sus asociados, a la vez que controlar toda la actividad de la empresa. El problema es que es una aplicación antigua y está mal diseñada, dificultando tanto los procesos de negocio que muchas cosas de las que ofrece las acababan haciendo a mano. Por ello, solicitaron una aplicación nueva que sustituyera a la antigua, que arreglara los defectos de la misma y que implementara nuevas funciones.

Los servicios de la asociación son amplios: asesoría jurídica, prevención de riesgos, cursos de formación, etc. Con todo, el objetivo del proyecto es conseguir que la aplicación abarque todas las áreas que trata la empresa, y llegue a ser una herramienta útil en el día a día de todos los trabajadores de la asociación. El presente PFC sólo se basa en parte del desarrollo de la aplicación final, esto se debe a que es un proyecto de larga duración

(inicialmente se estimó que el desarrollo duraría un año) y, por lo tanto, en el momento de presentar este PFC el desarrollo no se ha finalizado.

4.1.2 Requisitos

En este apartado se definen los requisitos generales de la parte de la aplicación descrita en este proyecto. Los requisitos son los siguientes:

- Gestión de procedimientos judiciales.
- Gestión de demandas.
- Gestión documental.
- Agenda de eventos (juicios, llamadas, reuniones, etc.).
- 100% configurable sin tocar el código.
- Generación automática de informes.
- Auditoría de cambios.

Por otra parte, y para acabar, se puede hacer una abstracción general de lo que se necesita. En este caso, todo se resume en dos requisitos: se necesita un almacén de datos (con una interfaz amigable para modificar los mismos) y una serie de funciones que trabajen sobre los datos almacenados.

4.1.3 Decisiones iniciales

Tras recoger las características y los requisitos de la aplicación, se han tomado una serie de decisiones que marcarán el proceso de desarrollo. La decisión más importante es encontrar un software *open source* que se adapte al máximo a nuestras necesidades, para luego poder modificarlo según las necesidades del cliente.

Para tomar estas decisiones se tuvo en cuenta la experiencia de la empresa desarrolladora. En este sentido, la misma ya usaba con anterioridad una serie de aplicaciones (un CRM, un gestor documental, y herramientas de inteligencia de negocio), y se verificó previamente la viabilidad de las mismas dentro del proyecto. En este estudio, se determino que el CRM (sugarCRM) utilizado por la empresa reunía las características requeridas (ver apartado **4.1.4**), por tanto, se estimó como una buena opción como base para el desarrollo del proyecto.

En cuanto al gestor documental utilizado (Alfresco, ver **3.3.3**), se vio que, gracias a sus múltiples vías de acceso a los documentos, daba bastantes facilidades a la hora de integrarlo con otras aplicaciones, con lo cual se decidió que sería útil para solucionar el requisito de gestión documental. También hay

que destacar sus grandes posibilidades de personalización como un factor que ayudó a la decisión.

Por otro lado, se necesitaba una aplicación que nos permitiera solucionar la parte de generación de informes. Para ello se estudio una de las herramientas de inteligencia de negocio utilizadas en la empresa, *Pentaho BI* [24]. Entre sus muchas funciones, esta aplicación cuenta con un motor de generación de informes muy potente el cual, además, puede ser accedido a través de servicios web, con lo cual se presenta como una solución ideal para integrarlo con las otras aplicaciones.

Otras decisiones deberían ser el lenguaje de programación a usar, el entorno de desarrollo y el tipo de aplicación a desarrollar (de escritorio, aplicación web, etc.), pero al partir de un software existente todos estos aspectos ya están decididos por sus creadores originales.

Con todas las decisiones tomadas, se paso al siguiente paso, es decir, a una nueva reunión con el cliente para acabar de aclarar los requisitos generales y empezar a concretar los requisitos concretos del primer departamento a desarrollar: el de asesoría jurídica.

4.2 SugarCRM

Tal y como se ha dicho en el apartado anterior, como aplicación *open source* a modificar se ha elegido SugarCRM. En este apartado se describen las principales características de esta aplicación, así como los motivos por los cuales se ha elegido esta aplicación por encima de otras.

El principal aspecto de esta aplicación no se trata de una característica, sino de la filosofía de desarrollo de la misma. Esta filosofía se basa en un desarrollo abierto, basado en la comunidad de usuarios de la aplicación por encima de un equipo cerrado de desarrolladores. Gracias a distribuir mediante una licencia de código abierto, la comunidad de usuarios puede ayudar al desarrollo aportando nuevas ideas y, sobre todo, nueva funcionalidad; y más importante que eso, es que desde el equipo de desarrollo de SugarCRM se incita a desarrollar a los usuarios.

En este sentido, han puesto a disposición de la comunidad de usuarios una página web [25], en la cual los usuarios pueden subir nuevos módulos o modificaciones de los existentes para compartirlos con otros usuarios. Además, en la misma web hay un foro de discusión y una extensa wiki de documentación, todo para favorecer el desarrollo y la evolución de la comunidad.

Todo lo explicado es especialmente útil en el caso del presente proyecto, puesto que, además de la funcionalidad incluida “de serie”, se pueden solucionar otras necesidades con la ayuda de la comunidad, tanto en el caso de encontrar alguna característica requerida que ya haya sido implementada

por otros usuarios, como en el caso de tener alguna duda que la gente del foro o la propia wiki nos pueda resolver rápidamente.

Por otro lado, no hay que restar importancia a las características incluidas por defecto, las cuales de por si componen una aplicación muy completa y, sobre todo, intuitiva. De las características por defecto, las más interesantes son las siguientes:

- Páginas de inicio personalizables por los usuarios.
- Calendario interactivo y compartido.
- Gestor de actividades (las cuales se reflejan en el calendario del usuario a quien han sido asignadas).
- Cliente de e-mail desarrollado en AJAX.
- Cuadro de mando personalizable.
- Agenda de clientes y contactos.
- Gestión de distintos aspectos relacionados con los clientes: Oportunidades, campañas de marketing, gestión de incidencias, etc.
- Se puede usar como portal para acceder a otras aplicaciones haciendo uso de iFRAMES.
- Lector de fuentes RSS.

También es interesante hablar sobre el diseño de la aplicación y, sobre todo, de sus módulos. En cuanto a estos últimos, SugarCRM dispone de dos tipos principales: el primero de ellos sería los módulos de datos, los módulos de este tipo se componen, principalmente, de una clase que define los campos de datos que se van a guardar en cada registro de ese módulo, y de una serie de vistas para visualizar y editar dichos registros de datos; el otro tipo de módulo sería aquel que no guarda datos, sino que dispone de algún tipo de funcionalidad extra (un ejemplo sería el módulo del calendario, que no guarda nada, sólo muestra datos de otros módulos).

Teniendo en cuenta los tipos de módulos que se pueden incluir a SugarCRM, se puede abstraer que la aplicación no es más que un almacén de datos con una serie de herramientas que proporcionan la habilidad de gestionar esos datos de forma cómoda. Este es uno de los aspectos más importante que ha llevado a la elección de SugarCRM como aplicación base (si se mira el apartado de requerimientos, **4.1.2**, se puede ver que esto es justo lo que se busca).

En resumen, se ha elegido esta aplicación como base principalmente por tres aspectos: el primero es porque el perfil de la aplicación cumple con los requisitos principales (almacén de datos + herramientas de tratamiento de los

mismos), el segundo por su facilidad a la hora de añadir nuevas características y, el tercero, por la experiencia previa de la empresa desarrolladora con esta aplicación.

4.3 Desarrollo del proyecto

Como ya se ha comentado este PFC se centra principalmente en el seguimiento de la metodología, puesto que es el tema central del mismo. Por ello, el subapartado principal es el 4.3.1 que trata de la planificación y el seguimiento de toda esta etapa. El resto de subapartados trata de aspectos generales del desarrollo en si mismo, tales como el procedimiento seguido para diseñar las características a implementar, la forma en que se ha implementado dichas características o los resultados obtenidos al final del proyecto.

4.3.1 Seguimiento

En este apartado se muestra la planificación inicial del proyecto (en forma de diagrama de Gantt), y los problemas surgidos para el cumplimiento de dicha planificación. En las siguientes figuras se puede ver el diagrama de la parte desarrollada hasta el momento, la primera imagen muestra el inicio del proyecto y el ciclo habitual que sigue un proyecto basado en la metodología diseñada (las iteraciones).

Como se puede ver en la siguiente figura, el proyecto se inició en la semana del 14 de Abril, programando las reuniones iniciales (descritas en el apartado **2.3**). Tras esto, la idea era realizar iteraciones de dos semanas compuestas por las etapas descritas en el apartado de la metodología elegida (aprendizaje, diseño, desarrollo y testeo, para luego acabar con una reunión con el cliente).

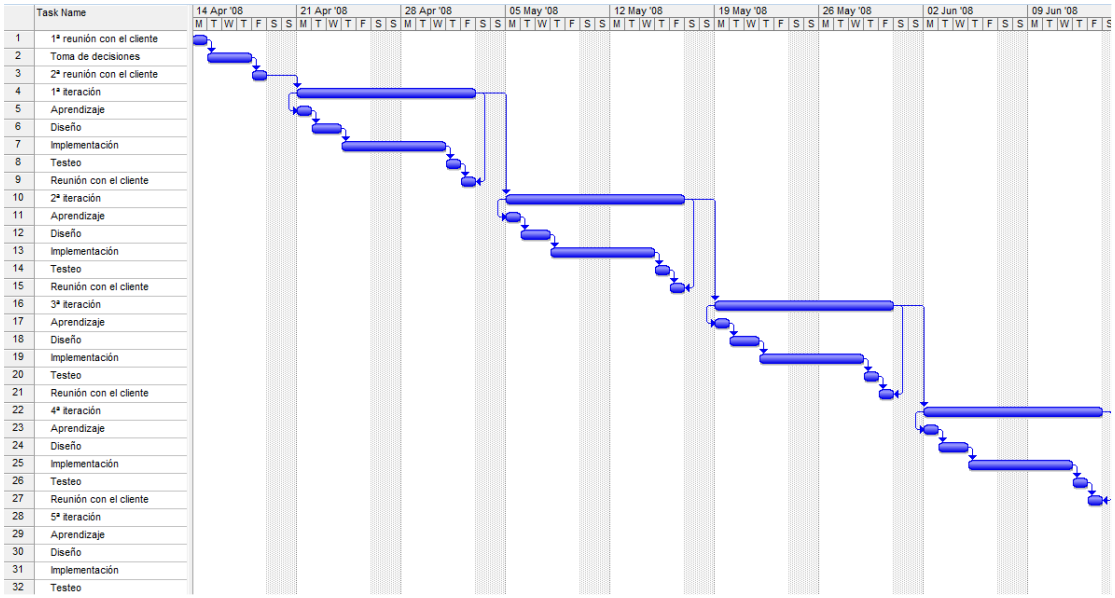

Fig. 4.1 Primera parte del diagrama de Gantt

Por otra parte, la segunda (y última) parte del diagrama de Gantt muestra el periodo estimado para el cierre de la parte de asesoría jurídica. Este se programó para el mes de agosto, ya que es el mes en el que la disponibilidad del cliente es menor (en este caso la disponibilidad en el mes de agosto era nula, por cierre vacacional de la asociación), por lo cual es el mejor momento para solucionar errores y mejorar el código existente (hay que recordar que, según la metodología diseñada, al finalizar un departamento se debe refactorizar el código y resolver errores). En la siguiente figura se puede observar la parte del diagrama correspondiente a lo explicado en este párrafo.

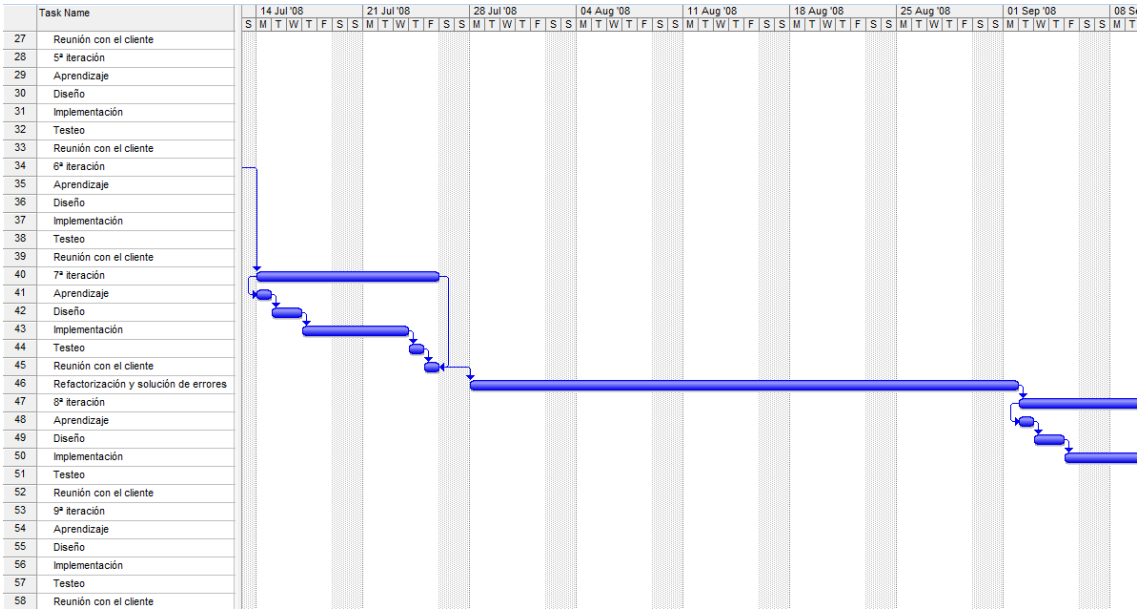

Fig. 4.2 Segunda parte del diagrama de Gantt

Hay que tener en cuenta que este diagrama de Gantt sólo se corresponde con la planificación inicial del proyecto. A la hora de la verdad no todo es tan fácil y es complicado cumplir los plazos (y aun más si se depende de alguien externo para avanzar). En el caso de esta metodología, la principal complicación es la disponibilidad del cliente (el mismo es necesario para la validación de los avances y para suministrar la información que haga falta), y en el proyecto que nos ocupa el cliente no ha estado siempre 100% disponible. Si bien es cierto que durante las primeras semanas del desarrollo se pudo mantener una reunión cada dos semanas y se pudo avanzar con paso firme, pero a medida que se avanzaba se fue dificultando la posibilidad de reunión, a la vez que se ralentizaba el avance del proyecto.

Otro problema importante que se ha presentado en el desarrollo del proyecto, es el hecho de que quien asistía a las reuniones con el cliente no era quien estaba desarrollando la aplicación, con lo cual las reuniones tampoco eran todo lo productivas que podían ser. Además de esto, se encuentra el agravante de que entre el desarrollador y el director no había comunicación cara a cara, sino por teléfono, con lo cual se dificulta el entendimiento, por parte del director, del

trabajo desarrollado, complicando a su vez la planificación de los temas a tratar en las reuniones.

Por otra parte, también ha habido algún caso de uso que se ha complicado más de la cuenta y se ha tenido que hacer en varias iteraciones, como es el caso de la parte del gestor documental, en la cual el tiempo de aprendizaje fue muy elevado a causa de que había que estudiar dos aplicaciones en vez de una sola y, además de esto, había que ver cómo integrar la una con la otra. Casos como este han hecho que se tenga que aplazar alguna reunión o que se haya hecho alguna con trabajo incompleto.

Aun con los problemas comentados (pertenecientes, sobre todo, a los últimos meses de desarrollo), la metodología ha sido bastante eficiente a la hora de desarrollar la parte de la aplicación correspondiente al departamento de asesoría jurídica, consiguiendo un buen grado de satisfacción por parte del cliente y, lo más importante, consiguiendo un entorno de gestión mucho más cómodo y flexible que el implantado con anterioridad.

Para acabar, de todo esto se puede sacar una conclusión general: la metodología ha sido bastante efectiva mientras el cliente ha estado disponible, pero se debe mejorar para que los momentos en los que no lo esté no se vuelvan improductivos.

4.3.2 Diseño

Las etapas (como define la metodología elegida, hay una en cada iteración) de diseño se han basado principalmente en el desarrollo de diagramas UML (ver **anexo A**) haciendo uso de la herramienta *Visual Paradigm* (apartado **3.1**). Lo primero que se hace es la definición de casos de uso a implementar en la presente iteración mediante diagramas de casos de uso. De esta forma, se han identificado las distintas características y procedimientos a implementar, así como los actores que intervienen en cada una de estas. En la siguiente figura se puede ver un ejemplo de definición de caso de uso tal y como se ha hecho en el proyecto.

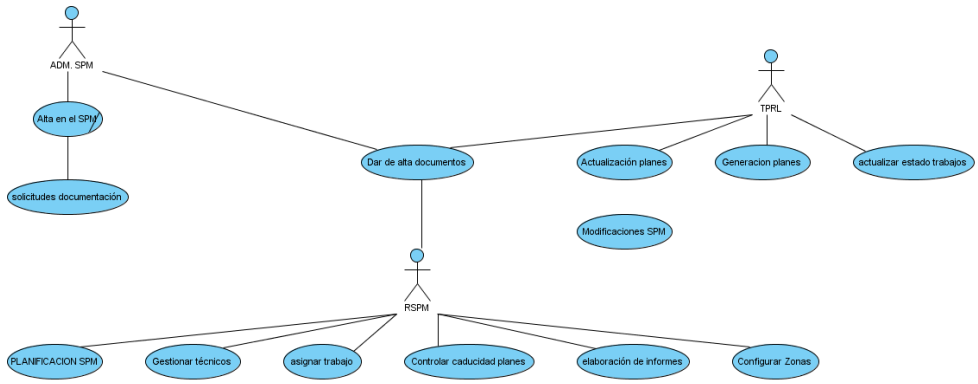

Fig. 4.3 Diagrama de casos de uso

Además del dibujo, *Visual paradigm* permite describir los casos de uso definidos. Para ello se debe hacer clic con el botón derecho sobre uno de los casos de uso y luego seleccionar la opción "*open specification*", para abrir la ventana de edición y luego rellenar el campo "documentación", situado en la pestaña "general". Este campo se ha rellenado para todos los casos de uso con la finalidad de describir los pasos de cada uno de ellos.

Una vez definidos los casos de uso, se procede a diseñar las características mediante diagramas UML. Por las características de la aplicación, la cual principalmente se trata de una aplicación de tratamiento de datos, lo primero es definir el modelo de datos, es decir, las distintas clases de objetos que se necesitan para implementar las características planificadas para la presente iteración. Para ello, lo mejor es usar un diagrama de clases, en el cual, además de los atributos aplicables a cada clase, se pueden ver las relaciones entre las distintas clases. La siguiente imagen muestra uno de los diagramas de clases creados.

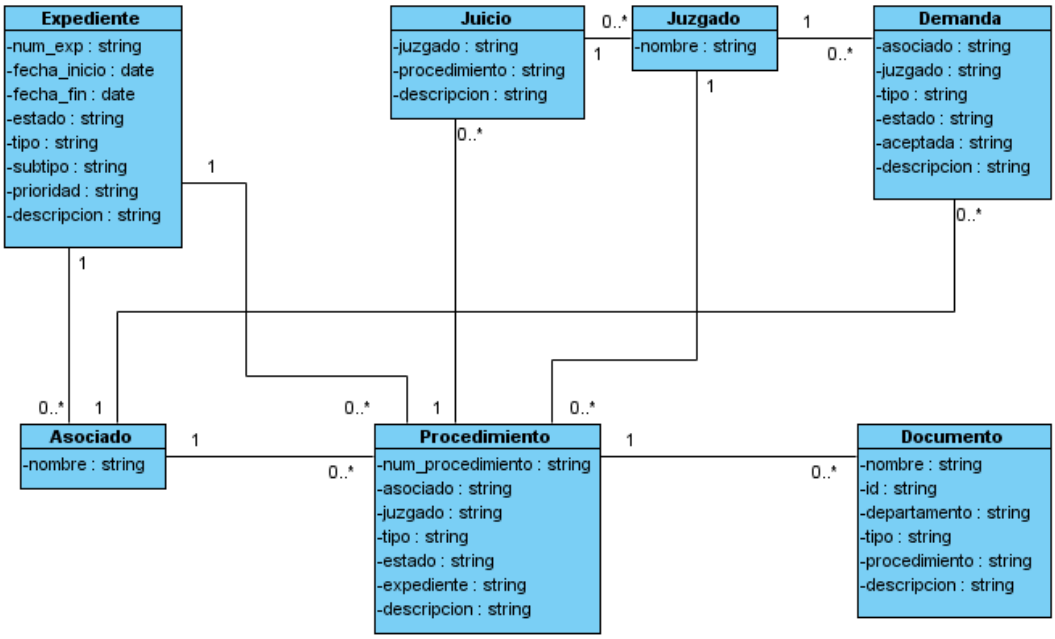

Fig. 4.4 Diagrama de clases de la parte de asesoría jurídica

El siguiente paso que se ha seguido es elaborar los diagramas de secuencia, de forma que se vea cómo interactúan los distintos componentes de la aplicación entre ellos a través de los procesos a implementar y ver los pasos que siguen dichos procesos. En este caso, se debe elaborar un diagrama por cada caso de uso que vayamos a implementar. En la siguiente figura se puede ver un ejemplo de este tipo de diagramas, correspondiente a un caso de uso concreto.

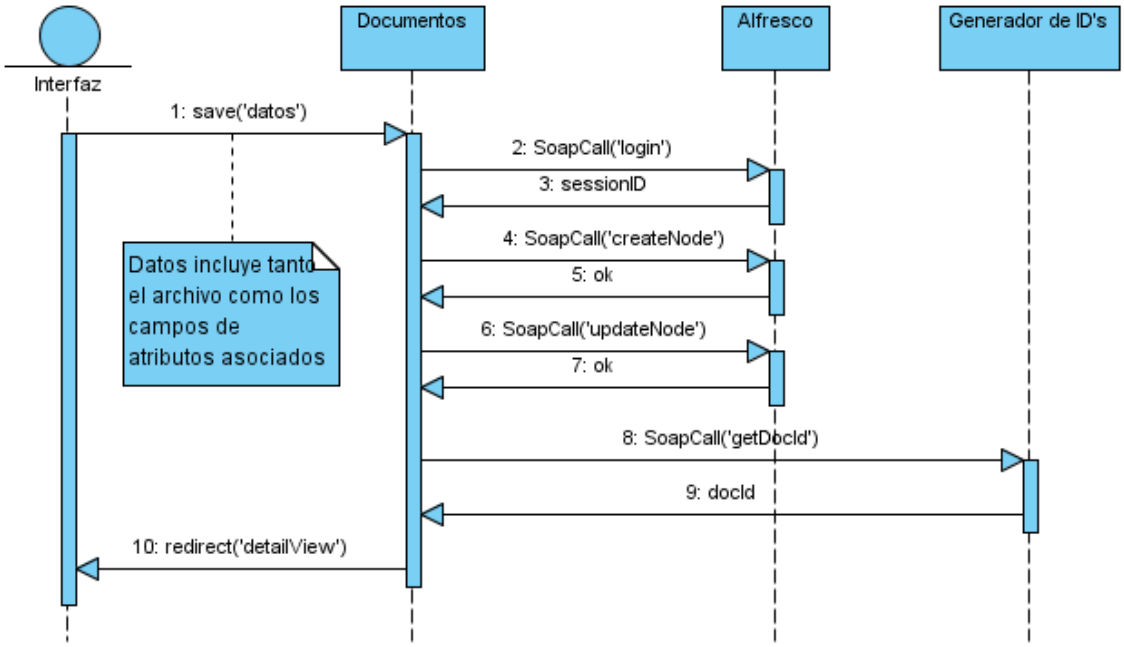

Fig. 4.5 Ejemplo de diagrama de secuencia

Una vez realizados estos diagramas, se dispone de toda la información necesaria para pasar a implementar las características requeridas en la presente iteración. Se podrían hacer más diagramas (ver apartado **3.1**), pero uno de los objetivos de la metodología diseñada es no gastar mucho tiempo en "papeleo" y pasar directamente a implementar características requeridas, por lo cual se estima que los diagramas descritos son suficientes para plasmar lo que se quiere hacer en cada iteración sin aumentar en exceso la carga de trabajo.

Finalmente, sólo queda comentar que, una vez realizados, los diagramas se suben al servidor de trabajo en equipo de la herramienta *Visual Paradigm* (*Teamwork server*), para poder ser visualizados y editados por todos los miembros del equipo de desarrollo.

4.3.3 Implementación

El corazón de cada iteración es el proceso de implementación de los casos de uso incluidos en dicha iteración, pero antes de eso hay que tener en cuenta la aplicación sobre la que se está trabajando. En este sentido hay que comprobar varias cosas, lo primero es ver si alguna función de la aplicación puede servirnos. En este caso, la aplicación (SugarCRM) es muy rica en funcionalidad y por lo tanto es fácil que se pretenda hacer algo que ya está hecho (aun más teniendo en cuenta los módulos desarrollados por la comunidad de usuarios, que también pueden ser de utilidad).

Si no se encuentra nada aprovechable, hay que hacerse la siguiente pregunta: a partir del diseño realizado, ¿Qué tipo de cambios son necesarios para implementar las características requeridas? En la mayoría de los casos, se ha determinado que son necesarios cambios de tres tipos: el primero sería introducir modificaciones en el modelo de datos; el segundo incluir lógica de negocio adicional; y el último modificar aspectos de la interfaz gráfica.

Si se mira el apartado en el que se definió la metodología a usar en este caso real, se puede ver que se habían predicho dos tipos de cambios: funcionales y estructurales. De los tres tipos de cambio definidos en este apartado, el primero se correspondería con los cambios estructurales; el segundo se correspondería con los cambios funcionales; mientras que el tercero sería un tipo de cambio que no se había contemplado: cambios visuales.

Otro aspecto que es interesante determinar es ver si la aplicación base dispone de algún mecanismo para realizar los tipos de cambio requeridos. En el caso de sugarCRM hay que decir que dispone de mecanismos para facilitar la implementación de los tres tipos de cambio enumerados. Para empezar, la aplicación (desde su versión 5.0) lleva incorporada una herramienta de creación de módulos, que permite crear módulos basados en objetos de datos persistentes, relacionarlos entre ellos, editar las diferentes vistas (edición, detalles y listado), modificar los campos de datos que contendrá, etc.

Esta característica, junto con el “estudio” (otra herramienta de edición incluida en SugarCRM, es similar al constructor de módulos pero permite editar los módulos de datos ya desplegados en la aplicación), permite realizar los cambios necesarios en el modelo de datos (crear nuevos objetos de datos, añadir campos a los ya existentes, etc.), además de permitir la edición de algunos aspectos de la interfaz gráfica (principalmente las vistas de los objetos de datos). En la siguiente imagen se puede ver el aspecto del constructor de módulos.

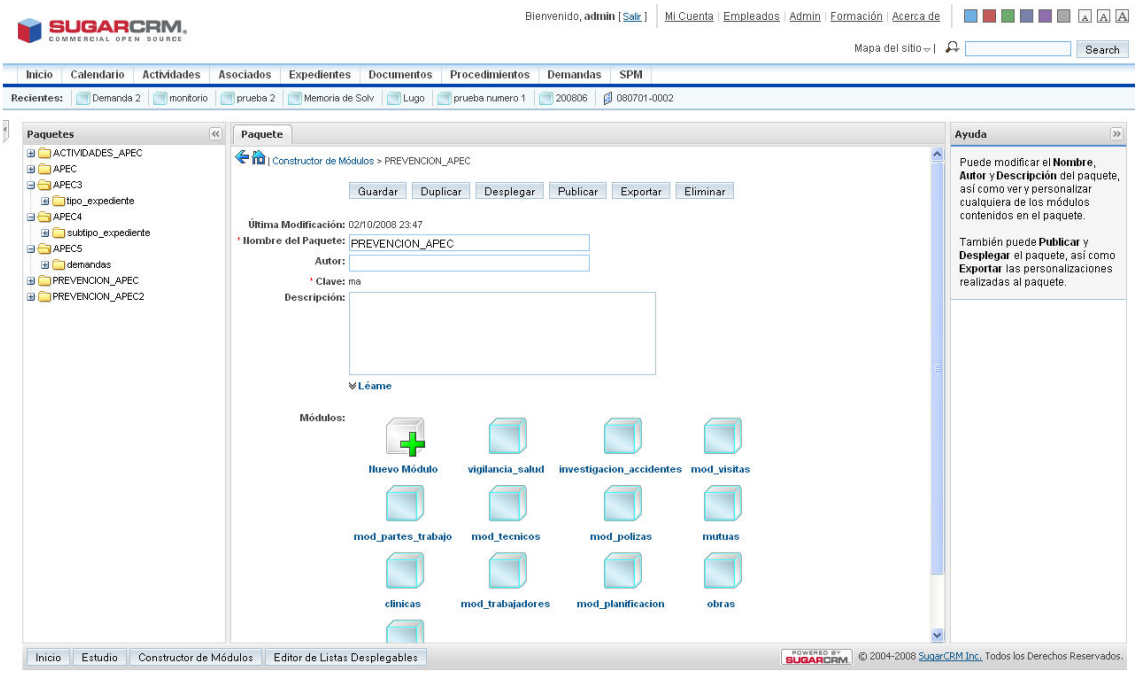

Fig. 4.6 constructor de módulos

Por otro lado, para añadir lógica adicional a los módulos de datos SugarCRM ofrece un mecanismo conocido como *"logic hooks"*. Este mecanismo permite ejecutar código fuente en determinadas condiciones de acceso a los objetos de datos almacenados. Por ejemplo, se puede hacer que se ejecute código justo antes de guardar un objeto después de editarlo, pudiendo generar el contenido de algunos campos de forma automática o realizar cualquier operación que sea necesaria en ese momento. Su funcionamiento es muy sencillo, simplemente hay que editar un archivo PHP, situado dentro del directorio del módulo al que se le quiera incluir la lógica adicional, en el cual se especifica los *"hooks"* que se quieren lanzar, cuando, en qué orden, y donde está situado el código a ejecutar (especificando archivo, clase y método de la clase), del resto se encarga SugarCRM.

La ventaja de este mecanismo, frente a la edición directa del código fuente original de la aplicación, es que el código que ejecutas los puedes poner donde quieras, con lo cual te permite tener controlada toda la lógica adicional que se va incluyendo en la aplicación, facilitando el control de cambios.

Todo lo explicado hasta ahora hace relación a modificaciones hechas sobre SugarCRM, pero en este proyecto dicha aplicación, usada como base, no trabaja sola, sino que se integra con otras aplicaciones. Como se ha dicho en el apartado **4.1.3,** junto con el CRM se integran las aplicaciones *Alfresco* (para solventar el requisito de gestión documental) y *Pentaho* (para la generación de informes). El mecanismo general que se ha usado para la integración son los servicios web. Hay que destacar que tanto SugarCRM como *Alfresco* y *Pentaho*, implementan el acceso a parte de su funcionalidad a través de servicios web, por lo cual se estimó que era la mejor forma de realizar la integración.

En el caso del gestor documental, se ha creado un módulo de SugarCRM que, en el formulario de entrada de nuevos registros, permite subir un archivo. Tras esto, antes de guardar el registro, se lanza un *hook* que, mediante servicios web, sube el archivo al servidor *Alfresco*. Aparte de esto, en el propio *Alfresco* se ha tenido que configurar el modelo de datos para definir los metadatos asociados a los documentos. Por otra parte, se han definido una serie de reglas de contenido que, según la información almacenada en los campos de metadatos, mueve el archivo a un directorio u otro, manteniendo un determinado orden de almacenamiento.

Para la generación de informes se barajaron dos posibilidades. La primera y más fácil fue usar las opciones de portal que incluye SugarCRM para acceder directamente, mediante HTTP, al servidor de inteligencia de negocio y pedirle los informes en formato HTML; esta solución es rápida de hacer ya que sólo hay que poner el enlace al informe correspondiente en el módulo “iFrames” de SugarCRM, pero tiene la desventaja de que hay que autenticarse en el servidor de inteligencia de negocio cada vez que se pide un informe, lo cual provoca una incomodidad de cara al cliente. Por otra parte, la segunda opción, es hacer la petición del informe a través de servicios web, con lo cual el usuario no se tiene que preocupar de la autenticación (el código PHP se encarga de ella de forma automática). La solución elegida es obvia, puesto que la segunda corrige el principal fallo de la primera, sin incluir demasiada complejidad.

Como se ha podido ver en este apartado, se han adoptado distintos tipos de cambio según las necesidades de cada caso de uso. La forma de llegar a las soluciones comentada es fruto de las etapas de aprendizaje. También faltaría comentar el hecho de que aun con los medios y herramientas que propone la aplicación base, algunas características se han tenido que desarrollar “a mano”, es decir, editando directamente el código fuente de la aplicación (algo que siempre es complicado).

4.3.4 Resultado

Para finalizar este capítulo, sólo queda comentar el resultado al que se ha llegado hasta el momento. Actualmente se ha dado por finalizada la parte de la aplicación correspondiente al departamento de asesoría jurídica y se ha iniciado la implementación de las características necesarias para el departamento de prevención de riesgos laborales. En las siguientes líneas se intentará describir, de manera resumida, los objetivos conseguidos en los dos departamentos abordados hasta el momento.

La parte del departamento de asesoría jurídica tiene como objetivo principal la gestión de procedimientos judiciales y de todos los aspectos que rodean esta gestión (documentos, planificación de juicios, expedientes, etc.). En este sentido se han desarrollado una serie de módulos para SugarCRM que permitan realizar esta gestión. Dichos módulos son los siguientes:

- **Asociados:** este módulo es en torno al cual girarán el resto, y con esto no me refiero a la parte de asesoría jurídica, sino a todos los departamentos. En él se guarda toda la información de los asociados, que son quienes reciben los servicios.

- **Procedimientos judiciales:** modulo de datos central, en el se especifican los detalles (número de procedimiento, estado, documentación asociada, etc.) de los procedimientos judiciales.

- **Expedientes:** con este módulo se pretende gestionar toda la actividad de la asociación. Un expediente relaciona distintos tipos de acciones tomadas para resolver una situación concreta. Entre dichas acciones se pueden incluir procedimientos judiciales, de hecho todo procedimiento judicial debe pertenecer a un expediente (cuando se guarda un procedimiento, si no se le especifica un expediente, crea uno nuevo y se lo asigna).

- **Documentos:** su nombre lo dice todo, se encarga de mantener un registro de todos los archivos subidos a la aplicación. Permite subir nuevos archivos y definirles ciertos campos de meta-datos, los archivos subidos mediante este módulo se almacenan en el servidor *Alfresco* (en un espacio determinado según las reglas de contenido configuradas).

- **Juicios:** este modulo permite programar juicios. Una vez programados se muestran en el calendario de la persona a quien se le ha asignado. También se ha desarrollado un script PHP que recuerda, a través de e-mail, los juicios un día antes de que se produzcan.

- **Demandas:** gestión de las demandas que las empresas asociadas, pretenden hacer a terceros. Incluye una función para transformar las demandas en procedimientos judiciales una vez estas han sido admitidas por el juzgado.

Como se puede ver en la descripción de los distintos módulos que componen esta parte de la aplicación, hay acciones que se ejecutan de forma automática (subida de archivos a *Alfresco* o creación de expedientes, por ejemplo). Por otra parte, hay que destacar que varios de los módulos descritos son reaprovechables para otras partes de la aplicación, cosa que ahorrará trabajo futuro.

Con estos módulos se consigue solucionar parte de los objetivos de la parte de asesoría jurídica. Otra parte que falta por trata es el requerimiento de que sea 100% configurable sin tocar el código. Esto se consigue con la implementación de varios módulos más que permiten definir listados de valores que luego se usarán en campos de los otros módulos (en forma de listas desplegables). Ejemplos de estos son los módulos “tipos de procedimientos”, “juzgados”, “estado de procedimientos”, entre otros.

En cuanto a la parte de prevención de riesgos, actualmente sólo está diseñado e implementado el modelo de datos y el alta de los asociados al servicio de

prevención. En este caso el modelo de datos es mucho más complejo al contener más tipos de objetos de datos y diversas relaciones entre ellos. Por ello surgieron problemas con él, los cuales fueron, principalmente, que había relaciones que no estaban del todo claras y que hubo que consultar con el cliente. Por otra parte, en el momento de empezar el desarrollo de esta parte se añadió un nuevo desarrollador al proyecto, el cual tuvo que empezar de cero con SugarCRM (no tenía experiencia previa con esta aplicación), con lo cual se retrasó un poco el avance del proyecto (lo suficiente para recibir una clases aceleradas, y por teléfono, del entorno de la aplicación, cosa que no es nada fácil).

Todo lo mencionado describe el estado actual del proyecto, el cual se podría decir que, cuando se termine con el departamento de prevención de riesgos, estará al 50%, ya que el resto de departamentos traerán menos carga de trabajo que estos dos (o al menos así se estimo al inicio del proyecto).

CONCLUSIONES

Las conclusiones que se pueden sacar de todo este estudio son varias. La más importante sería el hecho de que, sobre el papel, todas las metodologías son buenas, pero cuando se ponen en práctica es cuando aparecen los problemas. Las metodologías basadas en la filosofía ágil son muy motivadoras y, bien seguidas, pueden hacer que el desarrollo de aplicaciones se haga de forma muy eficiente y favorecedora de cara al cliente.

Por otra parte, el principal problema que se ha detectado es que el proceso depende continuamente de la participación del cliente. En el momento en que este no está disponible, por el motivo que sea, no se puede avanzar y, por tanto, se produce un retraso en el proceso de desarrollo.

Para intentar solucionar este problema se ha propuesto la siguiente solución: a medida que se avanza en el proyecto ir escribiendo un *"backlog"*, en el cual se vayan anotando posibles mejoras, pruebas de concepto o nueva funcionalidad, de forma que los periodos de tiempo en los que el cliente no está disponible se puedan aprovechar para mejorar la aplicación, implementando los aspectos especificados en el *backlog*.

De todas formas, aun con los problemas surgidos a causa de la baja disponibilidad del cliente, la metodología ha sido bastante eficiente, y gracias a ella se ha conseguido un producto (o parte de él, ya que la aplicación no está terminada) satisfactorio desde el punto de vista de los usuarios finales.

En cuanto al uso de herramientas que apoyen la metodología, una vez visto el funcionamiento de la misma con las herramientas elegidas, se podrían haber elegido mejor. Por ejemplo, al trabajar a distancia se ha echado en falta un sistema de gestión de proyectos más completo, sobre todo teniendo en cuenta que el desarrollador no podía asistir a las reuniones con el cliente y apuntarse las tareas a realizar. Estaría bien, de cara a futuros proyectos que sigan esta metodología, que el director del proyecto, tras las reuniones actualizara las tareas a realizar en el gestor de proyectos (ya sea para cerrarlas, pedir cambios en alguna de ellas o añadir nuevas).

En esta línea, también sería útil, en la etapa de diseño, añadir un nuevo diagrama UML: el diagrama de componentes. Este diagrama es especialmente útil en los casos en los que se tiene que integrar varias aplicaciones, ya que te muestra una imagen estática de las relaciones que tienen dichas aplicaciones (modeladas como componentes) entre si. Creo que se debería considerar casi obligatorio a partir de ahora, teniendo en cuenta las mejoras que se quieren introducir en la implementación (las cuales están expuestas más abajo en estas conclusiones).

Por otra parte, la principal conclusión a la que se llega acerca del desarrollo del caso real es que, si bien la solución de los *logic hooks* es bastante efectiva a la hora de añadir nuevas funciones a la aplicación, también introduce mucha

complejidad al posterior mantenimiento de la aplicación. Esta complejidad viene dada, principalmente, por el hecho de que se añade código adicional sin ningún control, con lo cual si surge algún error inesperado puede ser complicado encontrar de donde viene dicho error.

Actualmente se está trabajando en una solución basada en servicios web y en un servidor de procesos (Intalio BPMS) y flujos de trabajo. En dicha solución toda la lógica adicional se desarrollaría como funciones de un servicio web, los cuales serían llamados desde el servidor de procesos. Dicho servidor permite diseñar (mediante el lenguaje de descripción de procesos de negocio BPL) procesos que llamen a funciones alojadas en servicios web, los cuales, a su vez, pueden ser arrancados con una llamada a un servicio web. La idea es que desde los *logic hooks* sólo se llame a las funciones del servicio web del servidor de procesos, el cual se encargará de hacer todas las operaciones necesarias para completar todos los casos de uso complejos.

Con esta solución se consiguen varias cosas: por una parte se mantiene un orden en las funciones que se van añadiendo; por otra siempre que se quiera se puede consultar los procesos de la aplicación para localizar las funciones que se van ejecutando; por último, pero no menos importante, los procesos sirven como puente para integrar varias aplicaciones que dispongan de interfaz por servicios web.

El problema que puede presentar la solución expuesta sería ver el impacto que tendría, en cuanto a rendimiento, el introducir una nueva aplicación externa (el servidor de procesos). Si el rendimiento se viese gravemente afectado, esta solución no podría ser llevada a cabo puesto que no compensarían las ventajas ofrecidas.

Como se puede ver, no todas las decisiones tomadas han sido acertadas. Por ello, y siguiendo otro de los principios de las metodologías ágiles, hay que ajustar la manera de hacer las cosas de cara a la recta final del desarrollo de la aplicación en curso, eso sí, siempre con la ventaja de que, desde el punto de vista ágil, que se tenga que hacer cambios no es algo negativo, sino una vía para mejorar el rendimiento del equipo de desarrollo.

El impacto medioambiental de este proyecto es prácticamente nulo. Todo el trabajo realizado se basa en software y en metodologías de software, por lo tanto lo único que se consume es la energía eléctrica necesaria para hacer funcionar las máquinas que ejecutan el software desarrollado.

BIBLIOGRAFÍA

- Libros:

[1] Sommerville, I., *Ingeniería del software*, Pearson educación S.A., Madrid (2005).
[2] Eriksson, H. E., Penker, M., Lyons, B., Fado, D., *UML 2 toolkit*, Wiley Publishing, Indianapolis (2004).

- Páginas web:

[3] http://www.agilemanifesto.org
[4] http://ww.visual-paradigm.com
[5] http://ant.apache.org
[6] http://maven.apache.org
[7] http://msdn.microsoft.com/es-es/vstudio/default.aspx
[8] http://www.zend.com
[9] http://www.netbeans.org
[10] http://www.eclipse.org
[11] http://ww.apache.org
[12] http://www.dokuwiki.org
[13] http://www.knowledgetree.com
[14] http://www.alfresco.com
[15] http://www.mediawiki.org
[16] http://www.dotproject.net
[17] http://www.project-open.com
[18] http://www.vmware.com
[19] http://office.microsoft.com/es-es/project/default.aspx
[20] http://www.openworkbench.org
[21] http://www.nongnu.org/cvs/
[22] http://subversion.tigris.org
[23] http://ww.sugarcrm.com
[24] http://www.pentaho.com
[25] http://www.sugarforge.org

ANEXOS

ANEXO A: Diagramas UML

En este anexo se pueden ver la descripción de los 11 tipos de diagramas UML que existen, así como un ejemplo de cada uno.

- **Diagrama de clases:** un diagrama de clases es un tipo de diagrama estático que describe la estructura de un sistema mostrando sus clases, atributos y las relaciones entre ellos. Los diagramas de clases son utilizados durante el proceso de análisis y diseño de los sistemas, donde se crea el diseño conceptual de la información que se manejará en el sistema, y los componentes que se encargaran del funcionamiento y la relación entre uno y otro. La siguiente figura muestra un ejemplo de este tipo de diagramas:

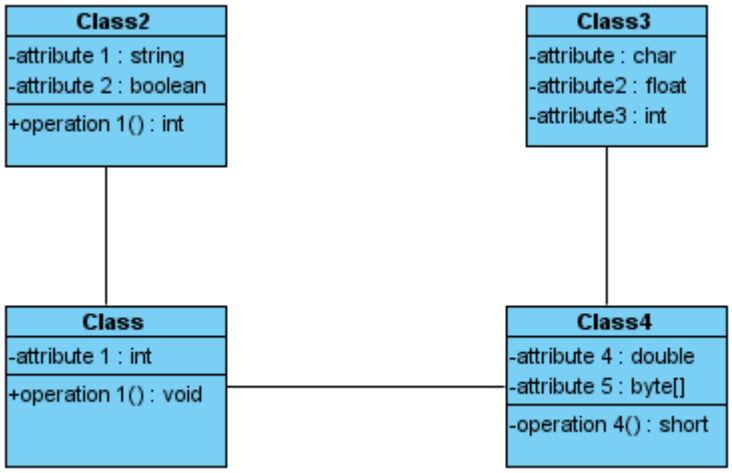

Fig. A.1 Diagrama de clases

- **Diagrama de componentes:** representa la separación de un sistema de *software* en componentes físicos (por ejemplo archivos, cabeceras, módulos, paquetes, etc.) y muestra las dependencias entre estos componentes. Se utilizan para modelar la vista estática del sistema. Nuevamente, se puede ver un ejemplo de estos diagramas en la siguiente figura:

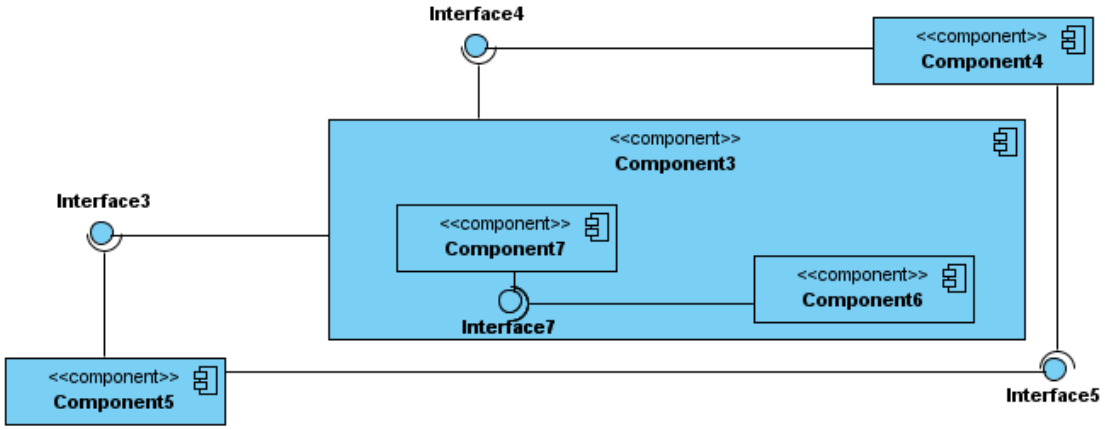

Fig. A.2 Diagrama de componentes

- **Diagrama de estructura compuesta:** este tipo de diagrama pretende definir aquellos objetos que están compuestos por otros. Se definen clases contenedor que están compuestas en su interior por otras clases. Por ejemplo, se podría tener una clase “coche” que esté compuesta por los objetos “motor”, “carrocería” y cuatro objetos “rueda”. Para ver un ejemplo de diagrama de estructura compuesta, ver la siguiente figura:

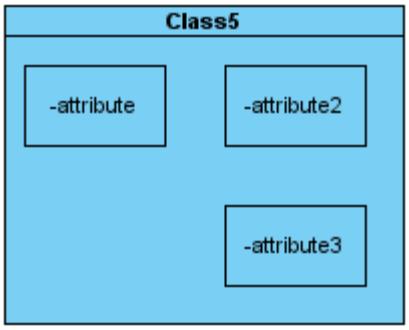

Fig. A.3 Diagrama de estructura compuesta

- **Diagrama de despliegue:** es un tipo de diagrama que se usa para modelar el hardware utilizado en la implementación del sistema. Principalmente permite especificar las características del hardware (SO, lenguaje de programación, máquina única o clúster de máquinas, etc.) donde funcionara cada componente del sistema, así como definir las relaciones entre dichos componentes. En la siguiente figura se puede ver un ejemplo de este tipo de diagrama:

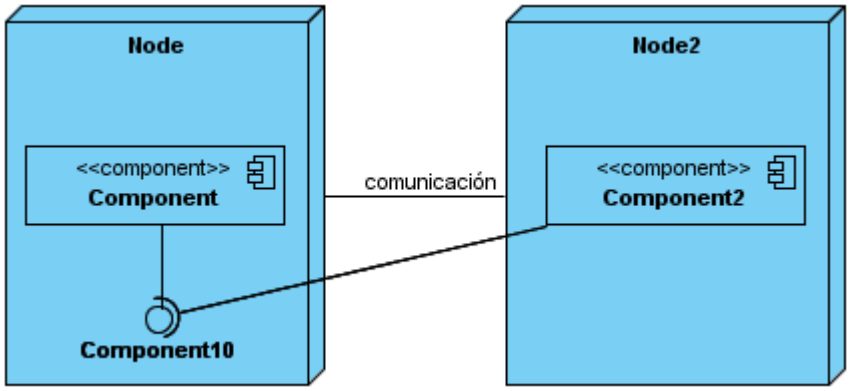

Fig. A.4 Diagrama de despliegue

- **Diagrama de actividades:** muestran el flujo de trabajo desde el punto de inicio hasta el punto final, mostrando los distintos caminos que se pueden seguir según el resultado de alguna actividad. Se puede decir que describe los pasos que se siguen para implementar un determinado caso de uso. La siguiente figura es un ejemplo de diagrama de actividad:

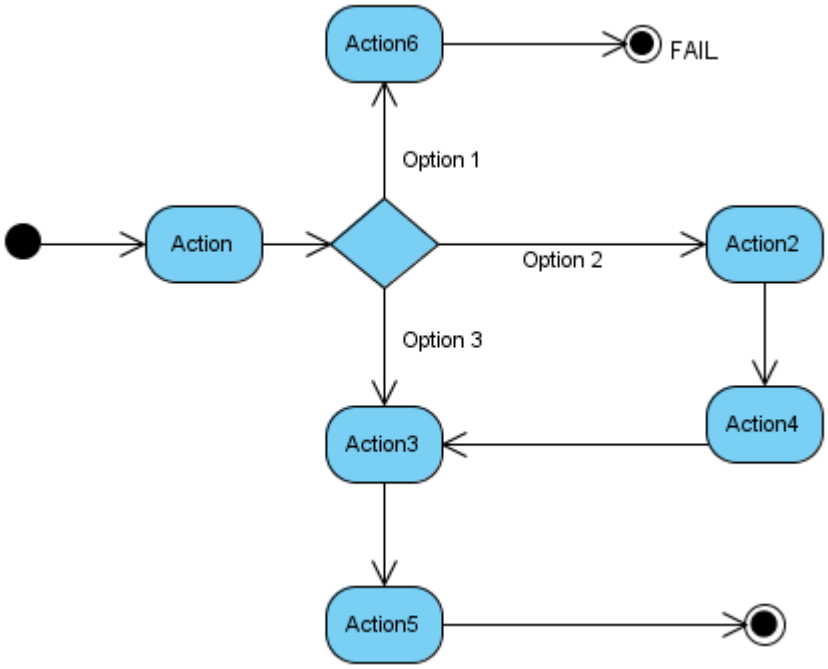

Fig. A.5 Diagrama de actividades

- **Diagrama de casos de uso:** son diagramas que se usan para representar de forma gráfica los casos de uso de un sistema. Los casos de uso se pueden definir como las acciones que puede (o que debería poder) tomar cada tipo de usuario en el sistema. En el diagrama se pueden ver los distintos actores del sistema representados por muñecos de palo señalando a las acciones que pueden tomar (representadas

mediante globos). La siguiente figura propone un ejemplo de diagrama de casos de uso.

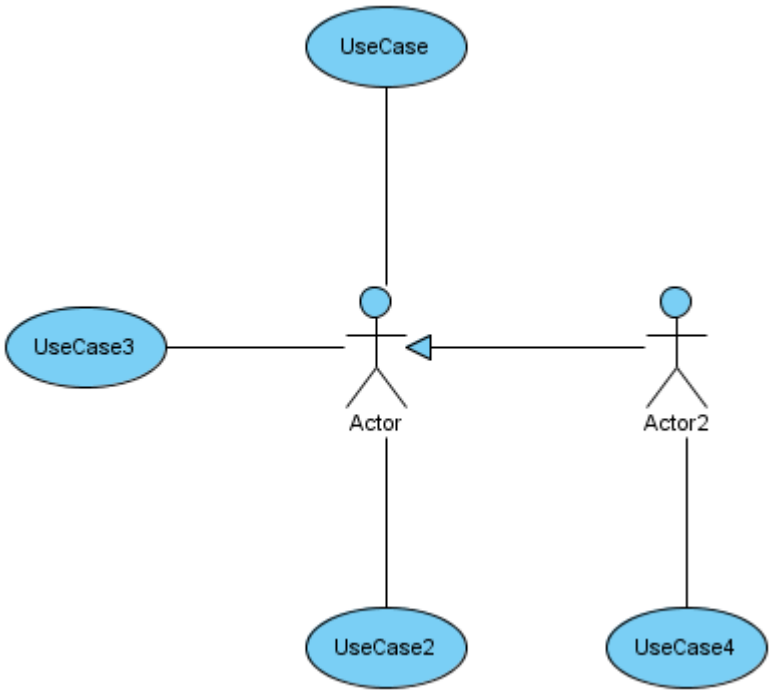

Fig. A.6 Diagrama de casos de uso

- **Diagrama de estados:** se utilizan para representar la máquina de estados del sistema que se está diseñando. En él se puede ver los distintos estados en los que puede encontrarse el sistema y las transiciones que hacen que el sistema pase de un estado a otro. En la siguiente figura aparece un ejemplo de diagrama de estados:

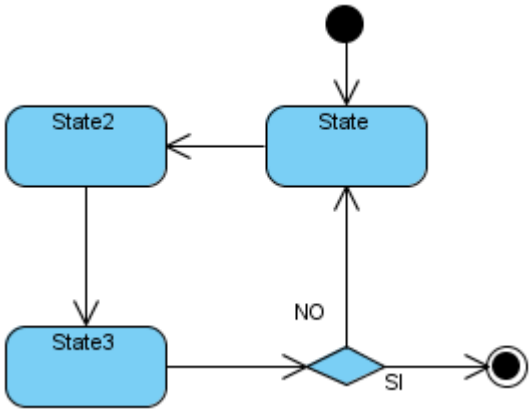

Fig. A.7 Diagrama de estados

- **Diagrama de secuencia:** El diagrama de secuencia es uno de los diagramas más efectivos para modelar interacción entre objetos en un sistema. Un diagrama de secuencia muestra los objetos que intervienen

en el escenario con líneas discontinuas verticales, y los mensajes pasados entre los objetos como vectores horizontales. Los mensajes se dibujan cronológicamente desde la parte superior del diagrama a la parte inferior; la distribución horizontal de los objetos es arbitraria. De esta forma se detallan los objetos y métodos que componen cada procedimiento de la aplicación. En la siguiente figura se ve un ejemplo de este tipo de diagramas.

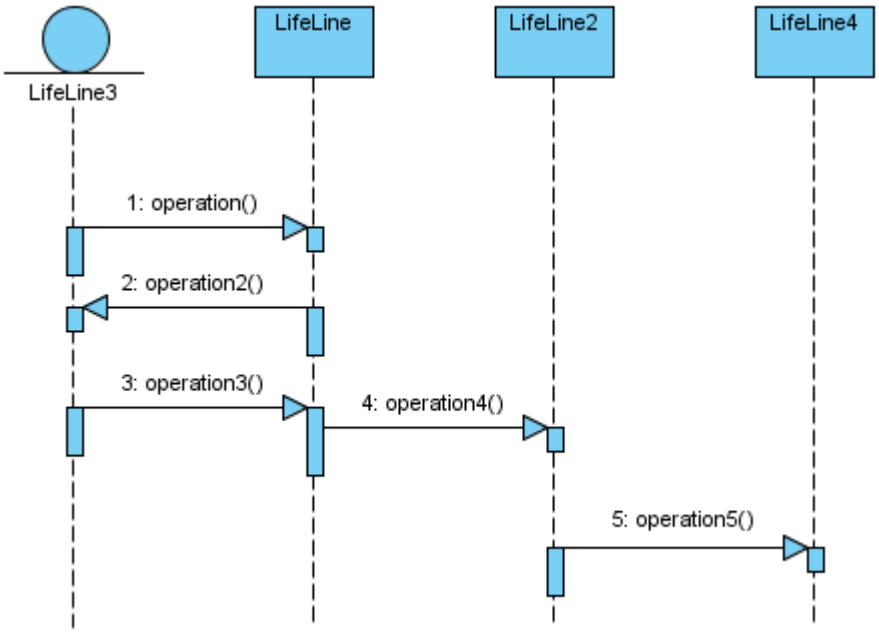

Fig. A.8 Diagrama de secuencia

- **Diagrama de colaboración:** es una alternativa al diagrama de secuencia. La diferencia principal con estos es que se centra en la relación entre objetos, sin mantener un orden cronológico de los mensajes entre estos. Por otro lado también muestra como las acciones entre los objetos van modificando las variables de cada uno. Para ver el aspecto de un diagrama de este tipo, ver la siguiente imagen:

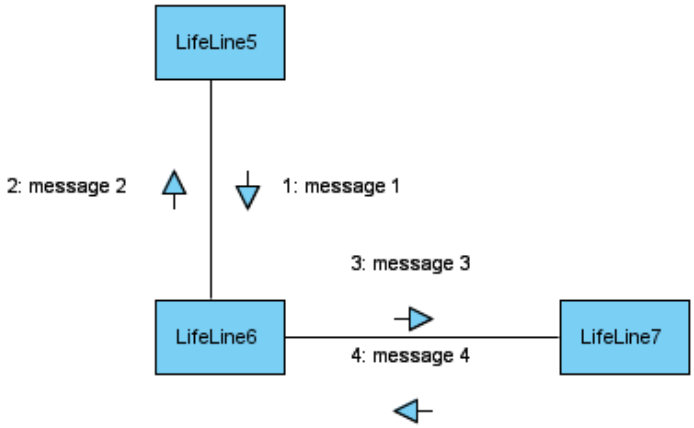

Fig. A.9 Diagrama de colaboración

- **Diagrama de tiempos:** Los diagramas de tiempos de UML se usan para mostrar el cambio en el estado o valor de uno o más elementos en el tiempo. Este también puede mostrar la interacción entre los eventos de tiempos, las restricciones de tiempos y la duración que los gobiernan. Se suelen usar este tipo de diagramas para modelar sistemas en tiempo real o embedidos.

- **Diagrama de vista de interacción:** muestra la interacción entre los objetos, está compuesto por un diagrama de secuencia más otro de colaboración. Como este diagrama está compuesto por dos mostrados anteriormente, no se muestra ningún ejemplo.